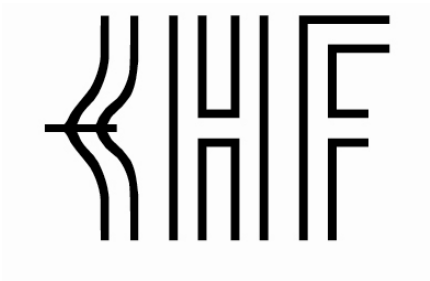

Institut für Hochfrequenztechnik

Universität Stuttgart

María de los Ángeles de la Cruz Barquero

Implementation of a Software Defined Ultra-Wideband Impulse Radio Receiver

Studienarbeit

Start Date: 20-12-2008

End Date: 18-08-2009

Supervisor: Dipl.-Ing. Marcel Blech

ABSTRACT

An implementation of a Software Defined Radio (SDR) Ultra-Wideband (UWB) Impulse Receiver is presented. The developed impulse receiver has been designed in order to be integrated in a Software Defined Impulse Radio (IR) Transceiver already implemented keeping the same quality rates.

The transceiver, where the receiver must be integrated, is implemented using Matlab language. Using Matlab language makes the execution time of the software slow. The new receiver implementation is carried out in C language in order to speed up the simulation. The speed increase of the simulation will offer a better performance for future different real time applications.

The method used for interact between Matlab and C language is a mex-function. The mex-function allows calling a C function from Matlab and transferring the data from Matlab to C and vice versa. The data is not transferred using files but they are transfixed as parameters using the memory. This practice allows speeding up even more the simulation.

The receiver prototype is characterized by being designed to work at a radio RF range of 2-3 GHz, as the transceiver. The impulse shape used to transmit the information exhibits a Gaussian shape and it has a -10 dB bandwidth of at least 500 MHz, as is required by UWB systems. The pulse repetition frequency established can oscillate between f_{sym} = 0.5 ... 62.5 MHz. The impulses shapes are modulated using a Binary-/Quadrature Phase-Shift Keying (B/-QPSK). The receiver is also designed employing bandpass (BP) sampling. The demodulation process is achieved using a matched filter and finally a maximum likelihood decision provides the digital received symbols.

Some measurements of the computational time needed to carry out the simulation for different data length are taken. They show that the solution proposed improves significantly the computational time, keeping the quality rates.

SYMBOLS

f_H [Hz]	Upper frequency of a signal
f_L [Hz]	Lower frequency of a signal
f_{sym} [Hz]	Symbol frequency
f_s [Hz]	Sampling frequency
f_0 [Hz]	Center frequency
$f_{s,BP}$ [Hz]	Bandpass sampling frequency
$f_{s,LP}$ [Hz]	Lowpass sampling frequency
T_s [s]	Sampling period
T_{sym} [s]	Symbol period
B [Hz]	Bandwidth
B_{BP} [Hz]	Bandpass bandwidth
B_{LP} [Hz]	Lowpass bandwidth
K [V]	Peak amplitude
g_t	Minimum amplitude of the baseband impulse
G_f [dB]	Attenuation at the corner frequency
T_f [s]	Period of equidistance symbols
Zc [Ω]	Characteristic impedance
σ_b	Variance value of the transmitted pulse train
μ_b	Mean value of the transmitted pulse train
τ [s]	Time delay parameter
θ_{PSK} [rad]	Angle of the PSK modulation
φ [rad]	Phase modulation constant
SNR [dB]	Signal to noise ratio

SIGNALS LIST

$g(t)$	Time domain baseband impulse
$G(f)$	Frequency domain baseband impulse
$g_s(t)$	Time domain sampled baseband impulse
$G_s(f)$	Frequency domain sampled baseband impulse
$g_m(t)$	Time domain modulated impulse
$h_{RF}(t)$	RF mode impulse response
$H_{RF}(f)$	RF mode Frequency response
$m(t)$	Time domain carrier signal
$g_{m,PSK}(t)$	Time domain Phase Shift Keying modulated signal
$PSD(f)$	Time domain power spectral density signal
$g_{s,RX}(t)$	Time domain sampled received signal
$G_{s,RX}(f)$	Frequency domain sampled received signal
$g_{RX}(t)$	Time domain received signal
$h(t)$	Channel impulse response
$n(t)$	Additive White Gaussian noise
$g_{MF}(t)$	Time domain matched filter output signal
$h_{MF}(t)$	Matched filter template impulse response
$g_{DAC}(t)$	Time domain digital analog converter output signal
$G_{s,RF}(f)$	Frequency domain sampled RF mode output signal

ABBREVIATIONS

UWB	**U**ltra-**W**ideband
IR	**I**mpulse **R**adio
B-/QPSK	**B**inary and **Q**uadrature **P**hase **S**hift **K**eying
DAC	**D**igital **A**nalog **C**onverter
ADC	**A**nalog **D**igital **C**onverter
SDR	**S**oftware **D**efined **R**adio
DSP	**D**igital **S**ignal **P**rocessing
PSD	**P**ower **S**pectral **D**ensity
I	**I**n phase signal
Q	**Q**uadrature phase signal
FCC	**F**ederal **C**ommunications **C**ommission
SNR	**S**ignal to **N**oise **R**atio
AWGN	**A**dditive **W**hite **G**aussian **N**oise
RZ	**R**eturn to **Z**ero
NRZ	**N**on **R**eturn to **Z**ero
OFDM	**O**rthogonal **F**requency **D**ivision **M**ultiplex
MC-UWB	**M**ulticarrier **U**ltra-**W**ideband
I-UWB	**I**mpulse **U**ltra-**W**ideband
ADSL	**A**symmetric **D**igital **S**ubscriber **L**ine
RF	**R**adio **F**requency
IF	**I**ntermediate **F**requency

TABLE OF CONTENTS

1. INTRODUCTION

In general terms a communication system can be considered as a UWB communication system if their instantaneous bandwidth is many times greater than the minimum required to deliver particular information in the 3,1 to 10,6 GHz band [1]. The evolution of UWB communications systems has been marked by the performance of the Federal Communications Commission (FCC).

The FCC is a regulatory body that was established by the Communications Act of 1934 as the successor to the Federal Radio Commission. It is charged with regulating all non-federal government use of the radio spectrum (including radio and television broadcasting), and all interstate telecommunications (wire, satellite and cable) as well as all international communications that originate or terminate in the United States. It is also the responsible for carving the spectrum into narrow slices, which are then licensed to various users. This carve of the spectrum into narrow slices became the biggest enemy of UWB communications and relegated it to experimental work for a very long time.

However, in the last decades, a lot of scientists and corporations were studying this system in an experimental way. They realized that UWB systems provided a lot of advantages regarding narrowband systems. This rendered in many request to the FCC asking for permission to operate unlicensed UWB systems concurrent with the existing narrowband signals. As a result of this, the FCC began detailed investigations into indoor/outdoor UWB propagations modelling as well as the the effect of UWB emissions on existing narrowband systems. Finally, as a result of those studies, the FCC changed the rules in 2003 to allow UWB system operation in a broad range of frequencies and the first commercial system was installed in the same year.

A more precise definition of a UWB is given by the FCC. According to the FCC, UWB is a signal with either a fractional bandwidth of 20% of the center frequency or 500 MHz [2]. Fractional bandwidth is defined by the formula $2\ (f_H - f_L) / (f_H + f_L)$ where f_H represents the upper frequency of the -10 dB emission limit and f_L represents the lower frequency limit of the -10 dB emission limit.

Today is well known that UWB systems have some features that differentiate them from the conventional narrowband systems [2]:

- Large instantaneous bandwidth enables fine time resolution for network time distribution, precision location capability, or use as radar.
- Short duration pulses are able to provide robust performance in dense multipath environments by exploiting more resolvable paths.
- Low power spectral density (PSD) allows coexistence with existing users and has a Low Probability of Intercept.
- Date rate may be trade for PSD and multipath performance.

A graphical comparison of the Fractional Bandwidth of a narrowband and UWB communication system is depicted in the figure 1.1.

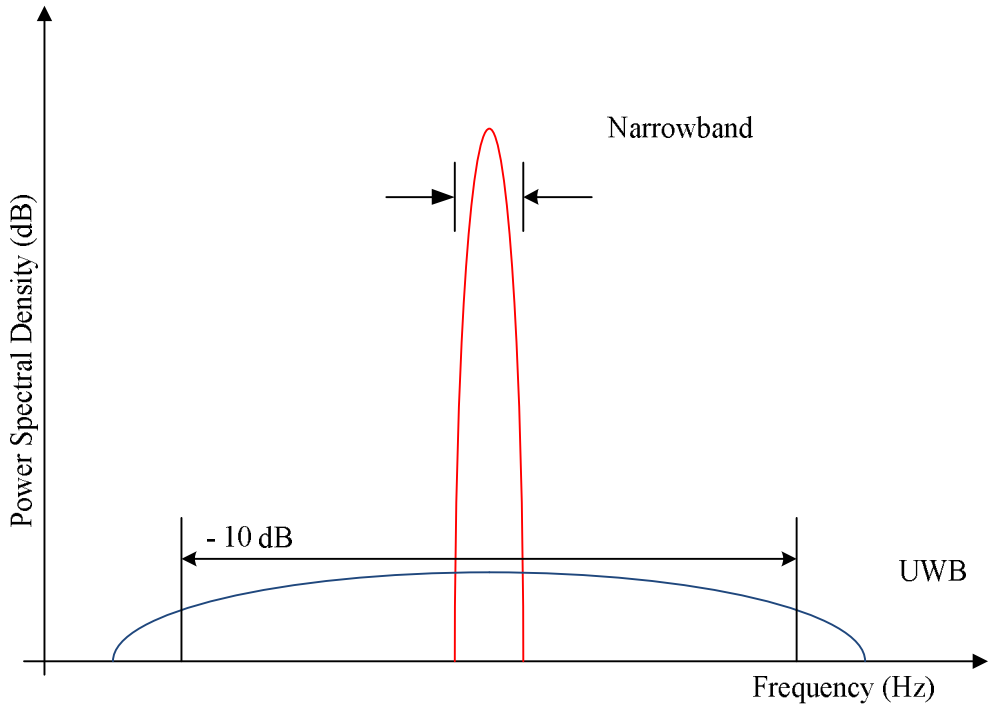

Fig. 1.1: Comparison of the Fractional Bandwidth of Narrowband and UWB communication system.

There are different forms of UWB signals [3]. Those forms are Impulse UWB (I-UWB) and Multicarrier UWB (MC-UWB). I-UWB is the kind of signal used to send the information in this research. I-UWB is based on sending very short duration pulses to convey information. This practice does not use a sinusoidal carrier to transfer the information but the transmit signal is a series of very short baseband pulses reaching bandwidth on the order of gigahertz. This signal can be represented as

$$s_{\mathrm{I}}(t) = \sum_{i=-\infty}^{\infty} A_{\mathrm{i}}(t)p(t - iT_{\mathrm{f}}), \tag{1.1}$$

where A_{i}(t) is the amplitude of the pulse equal to $\mp\sqrt{E_{\mathrm{p}}}$, where E_{p} is the energy per pulse and T_{f} is the frame repetition time.

The opposite signal form is MC-UWB which has its best exponent in Orthogonal Frequency Division Multiplexing (OFDM). OFDM is a technique that uses densely spaced subcarriers and overlapping spectra. The model of transmitted signal can be expressed as

$$s_{MC}(t) = \sum_{i=1}^{N} d_{\mathrm{i}}(t)e^{j2\pi i\left(\frac{t}{t_{sym}}\right)} \tag{1.2}$$

where N is the number or carriers, T_{sym} is the symbol duration, and $d_{\mathrm{i}}(t)$ is the symbol stream modulating the i^{th} carrier. Today, OFDM is a technique widely used every day in products like Asymmetric Digital Subscriber Line (ADSL), IEEE 801.11a/g...

MC-UWB has a very good efficiency avoiding interference because its carrier frequencies can be chosen to avoid narrowband interference to or from narrowband systems. They also provide more flexibility and scalability. On the other hand, IR systems increase the possibility of interference from UWB to narrowband systems due to the high instantaneous power during the brief interval of the pulse. IR signals require fast switching times for the transmitter and receiver precise synchronization. Simple I-UWB systems can be very inexpensive to construct.

As it was said, up to now mainly solutions based on OFDM are commercially available while IR devices are difficult to be founded. Despite this, IR is a very promising technology. For this, today has become a centre of investigations trying different techniques such as subsampling radio architecture [4], transmitted reference systems [5], [6].

The initial idea of this research is to design as the title says a software defined UWB IR receiver. The receiver must be integrated and compatible with the fundamentals of a UWB transceiver already implemented. Implementing a software simulation allows the possibility to make an approximation of the real results and consequently make a prediction with them. The results help to discover, develop, improve and discuss the convenience to make a hardware implementation of the prototype.

The entire transceiver has already been implemented by software. The platform chosen for this task was Matlab. Matlab provides a lot of tools for digital signals processing and this makes it the best option to carry out this development. The disadvantage of using Matlab is that requires more computational time than other languages and the need of a “real time” simulation makes it a not suitable option.

The main purpose of this study is to implement the aforementioned receiver using C language programming in order to get better computational times. The improvement of the computational times will be useful to make a “real time” simulation, but it must keep the same quality rates.

After meeting these assumptions, it is easy to understand the main objective of this research. The receiver programmed in C language must be able to intercommunicate with Matlab language because the other parts of the transceiver are implemented using it. Once this, the next step is to develop a receiver capable to receive the signals emitted in the transceiver. It must implement a B-Q/PSK demodulator and the whole system must works at the same frequencies.

2. FUNDAMENTALS

2.1 Transceiver

The Software Defined UWB IR Receiver presented was designed in order to satisfy the requirements of a Software Defined Subsampling UWB Binary and Quadrature Phase Shift Keying (B-/QPSK) IR Transceiver [7]. The transceiver has been already designed for a software and hardware implementation. The explanation is achieved from a theoretical point of view but always taking into account some hardware requirements.

The transceiver prototype is characterized by being designed to work at a radio frequency (RF) range of 2-3 GHz. The impulse shape used to transmit the information exhibits a Gaussian shape and it has a -10 dB bandwidth of at least 500 MHz as is required by UWB systems. The pulse repetition frequency established can oscillate between f_{sym} = 0.5 ... 62.5 MHz.

A traditional SDR transceiver would have a block diagram according to the figure 2.1 (top) but the transmitter shown in this section has a block diagram as the figure 2.1 (bottom).

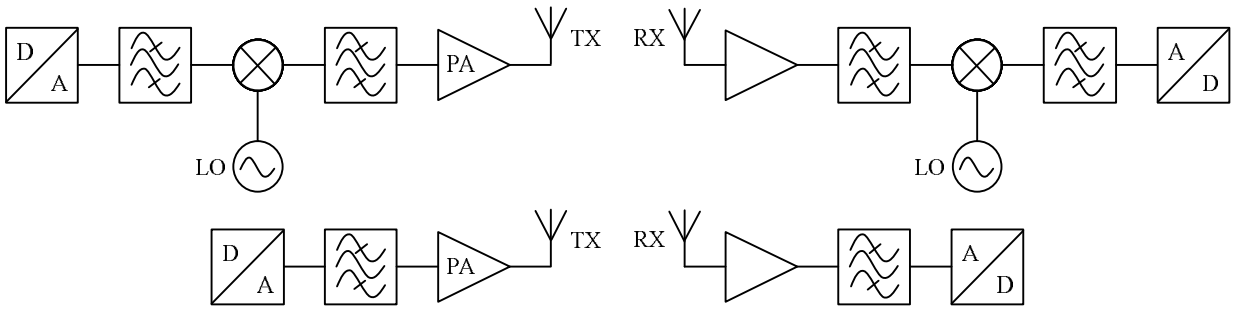

Fig. 2.1: Block diagram of a conventional SDR transceiver (top) and a subsampling architecture (bottom) [7].

As can be seen, the main difference between both prototypes is that the design presented has not a mixer stage like the traditional implementation. Conventional narrowband SDRs process the signal on an intermediate frequency (IF) and need additional mixer stages to make the up and down conversion to the RF. The addition of these mixer elements causes signal distortions due to their non-ideal transfer characteristic but it also helps to ensure a low noise level. Unlike the aforementioned solution, the presented system employs a subsampling architecture [8] without

mixers stages achieved by using bandpass sampling and a multi-Nyquist digital analog converter (DAC).

To summarize, bandpass sampling [9] is a sampling technique that allows decreasing the sampling rate. The classical sampling theorem states that a signal can be reconstructed if the sampling rate is at least $f_s = 2B$, where B is the signal bandwidth. This is a very good option when the signal spectrum is located at the origin of the coordinate axis, as can be seen in the figure 2.2.

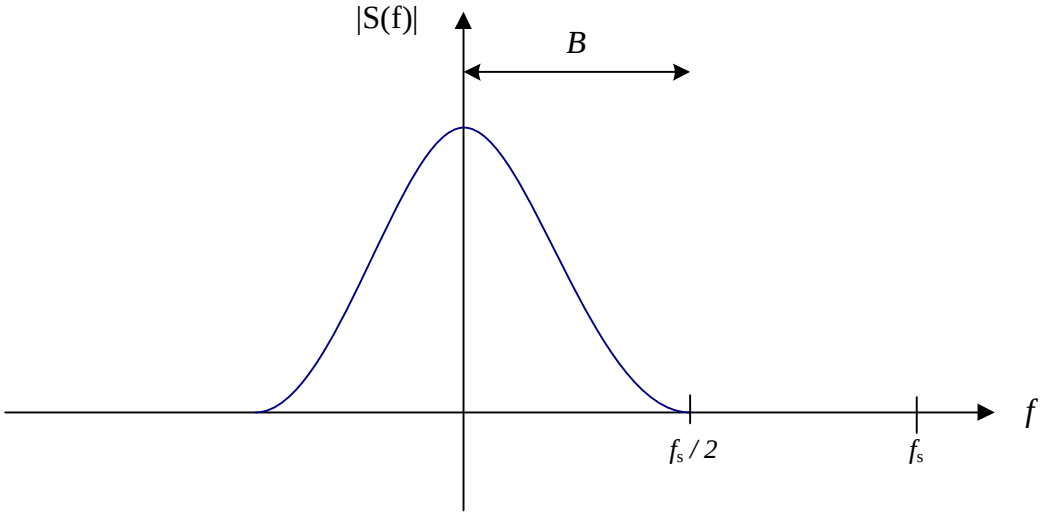

Fig. 2.2: Sampling situation with a signal spectrum at the origin of the coordinate axis.

The opposite situation would be when the signal spectrum is located at a very high frequency. In this case it would be necessary a very high sampling frequency and it is a requirement that sometimes is not possible to assume in hardware implementations. This situation is shown in the figure 2.3.

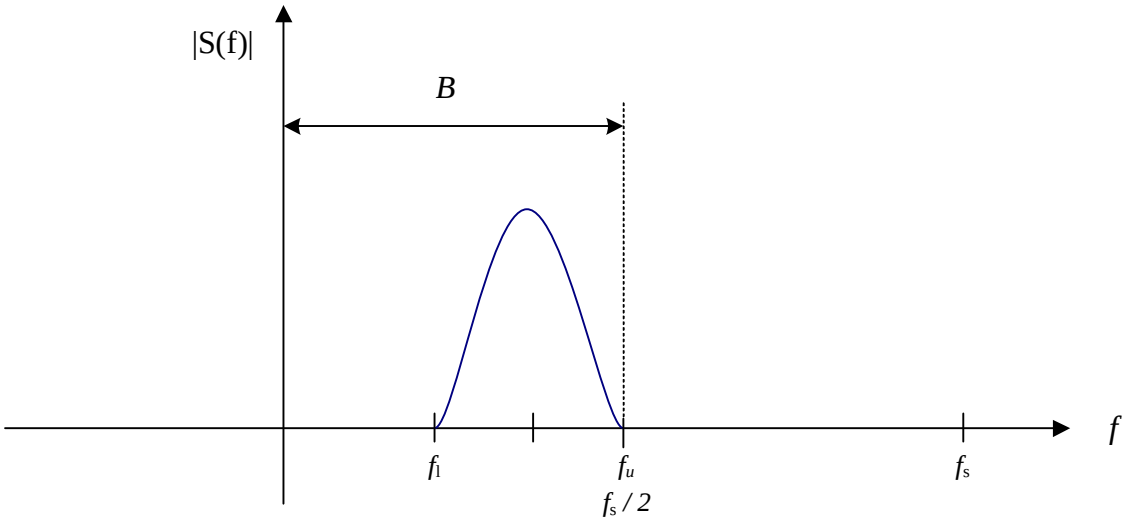

Fig. 2.3: Sampling situation with a signal spectrum at a very high frequency.

Bandpass sampling technique states that a signal previously filtered using a bandpass filter, can be reconstructed if the sampling rate is $f_{s,BP} = 2B_{BP}$, where $B_{BP} = f_u - f_l$. Using this technique is possible to reduce the sample rate in order to allow hardware implementations. The figure 2.4 evidences the results of applying traditional sampling and bandpass sampling. As can be seen, using traditional sampling it would be necessary a sampling frequency $f_{s,LP} = 2B_{LP}$, while using bandpass sampling it is only necessary a sampling frequency of $f_{s,BP} = 2B_{BP}$.

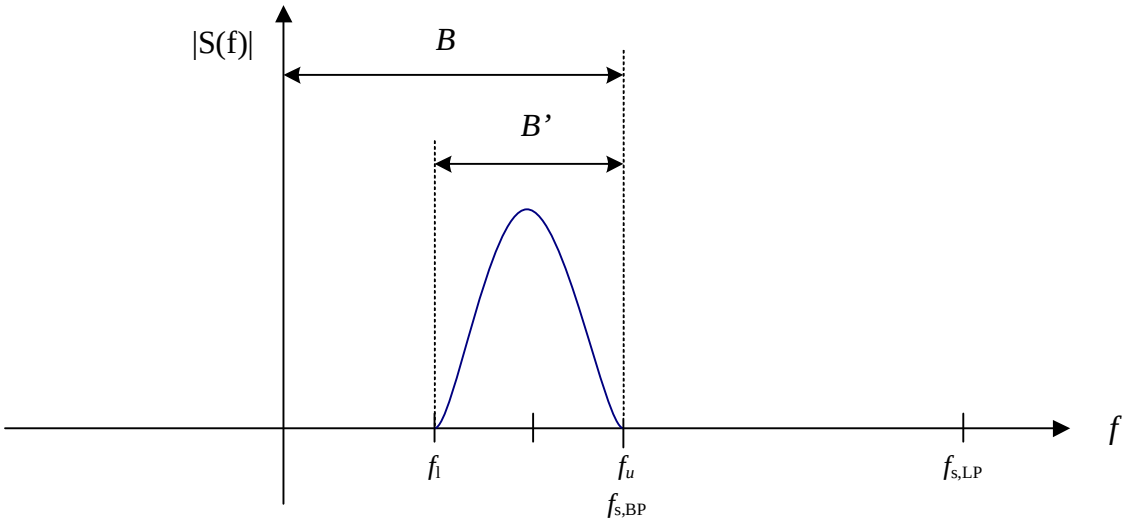

Fig. 2.4: Comparison of traditional sampling and bandpass sampling.

All the discrete signals have a periodic spectrum. If the signal is synthesized in the first Nyquist zone does also generate multiple attenuated replicas in the higher Nyquist zones. Taking advantage of this phenomenon, it is possible to use a bandpass filter in order to get directly the RF signal desired. This situation is illustrated in the figure 2.5. The disadvantage of this process is that sometimes, the signal replicas are much attenuated. To resolve this problem a multi-Nyquist DAC is employed. The aforementioned DAC allows obtaining replicas with acceptable amplitude. The figure 2.4 shows the difference between a normal DAC and a multi-Nyquist DAC outputs. It works with a sampling frequency f_s = 2 GHz making possible to work with signals with a bandwidth up to 1 GHz.

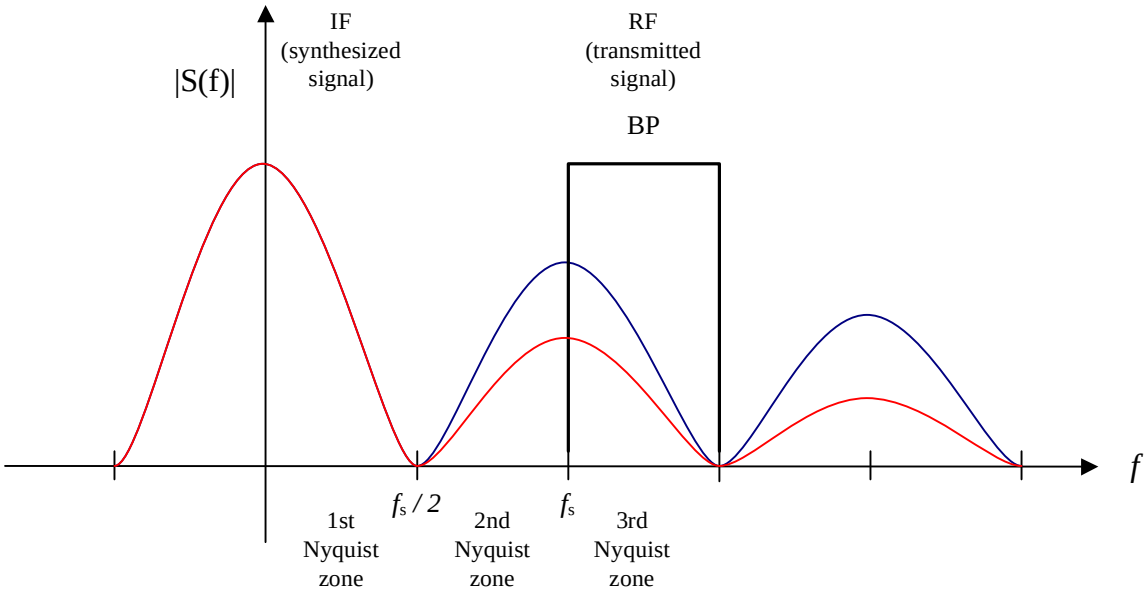

Fig. 2.5: Spectrum of a band-limited modulated sampled signals and the different DAC outputs.

The aforementioned DAC offers different output modes as is depicted in the figure 2.6. These modes are non return to zero (NRZ), the return to zero (RZ) and the RF mode. Each mode has different amplitude and application in the different Nyquist zones, thus, the RF mode provides the highest output amplitude in the third Nyquist zone. This DAC is not implemented in the signal processing of the software simulation.

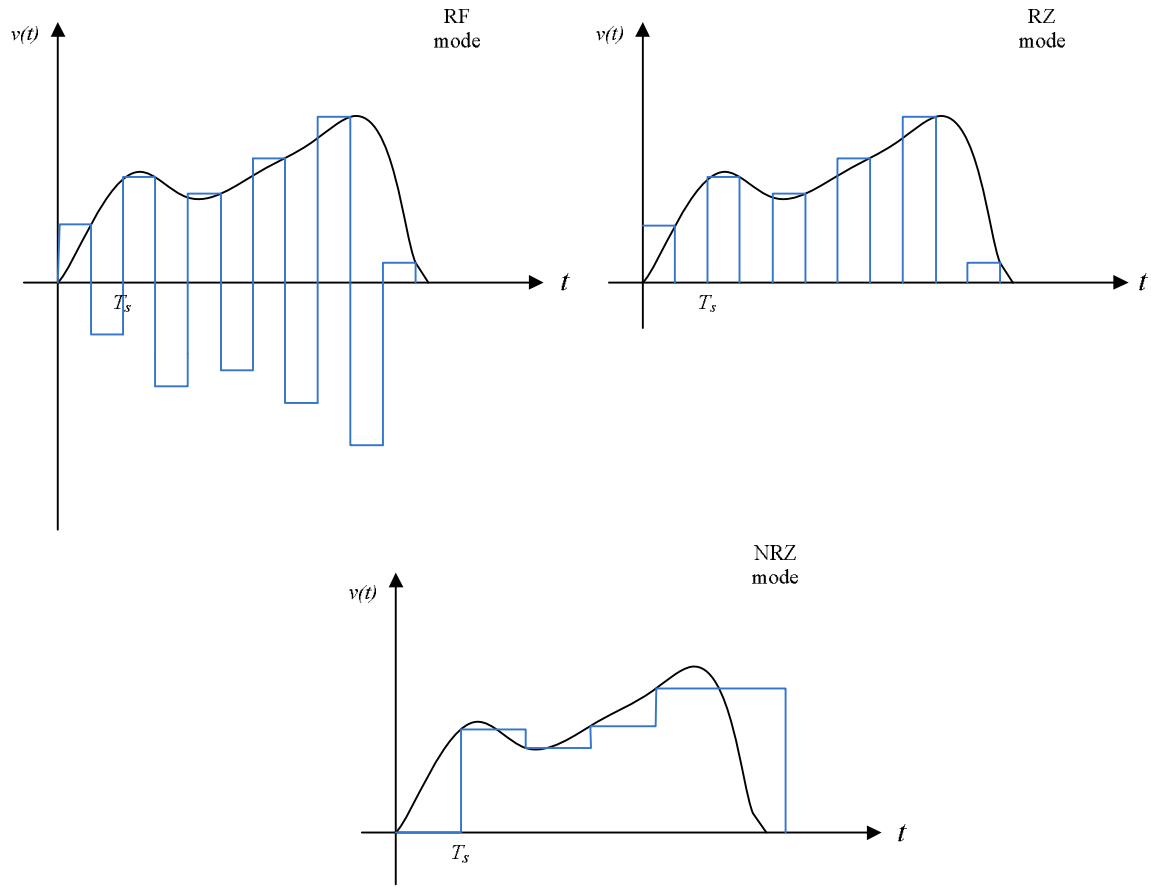

Fig. 2.6: Multi-Nyquist DAC output signals of a digitized analog waveform.

A single bipolar output impulse in the RF mode can be expressed as

$$h_{\mathrm{RF}}(t) = \mathrm{rect}\left(\frac{t+\frac{T_S}{4}}{\frac{T_S}{2}}\right) - \mathrm{rect}\left(\frac{t-\frac{T_S}{4}}{\frac{T_S}{2}}\right) \tag{2.1}$$

$$H_{\mathrm{RF}}(f) = jT_s\,\frac{1-cos(\pi f T_S)}{\pi f T_s}, \tag{2.2}$$

where T_s denotes the sampling interval, $h_{\mathrm{RF}}(t)$ represents the time domain waveform and $H_{\mathrm{RF}}(f)$ is frequency domain representation of the RF impulse. The transmitted pulse train $g(t)$ which is explained in the transmitter section is centred at a carrier frequency within the first Nyquist zone. The individual impulses $g(t)$ are sampled at the sampling interval T_s yielding the signal

$$g_s(t) = g(t)\sum_{n=-\infty}^{\infty}\delta(t-nT_s) \tag{2.3}$$

In order to obtain the spectrum of the DAC output signal $g_{\mathrm{DAC}}(t)$, the sampled pulse train must be convolved with the according impulse waveform of the chosen DAC mode.

$$g_{\mathrm{DAC}}(t) = h_{\mathrm{RF}}(t) * g_{\mathrm{s}}(t) \tag{2.4}$$

This yields the output spectrum

$$G_{\mathrm{s,RF}}(f) = H_{\mathrm{RF}}(f)\frac{1}{T_{\mathrm{s}}}\sum_{k=-\infty}^{\infty} G_{\mathrm{s}}\left(f - \frac{k}{T_{\mathrm{s}}}\right) \tag{2.5}$$

First order sampling is also used for this prototype. This solution offers a limit in the usable frequency range but relaxes the requirements of the DAC stage. On the contrary, second order sampling would offer more advantages respect to the usable frequency range but on the other side it needs more complex digital signal processing (DSP) [10]. The limit in the usable frequency range is caused for the need to synthesize the signal in the first Nyquist zone. The difference between both kinds of sampling is depicted in the figure 2.7.

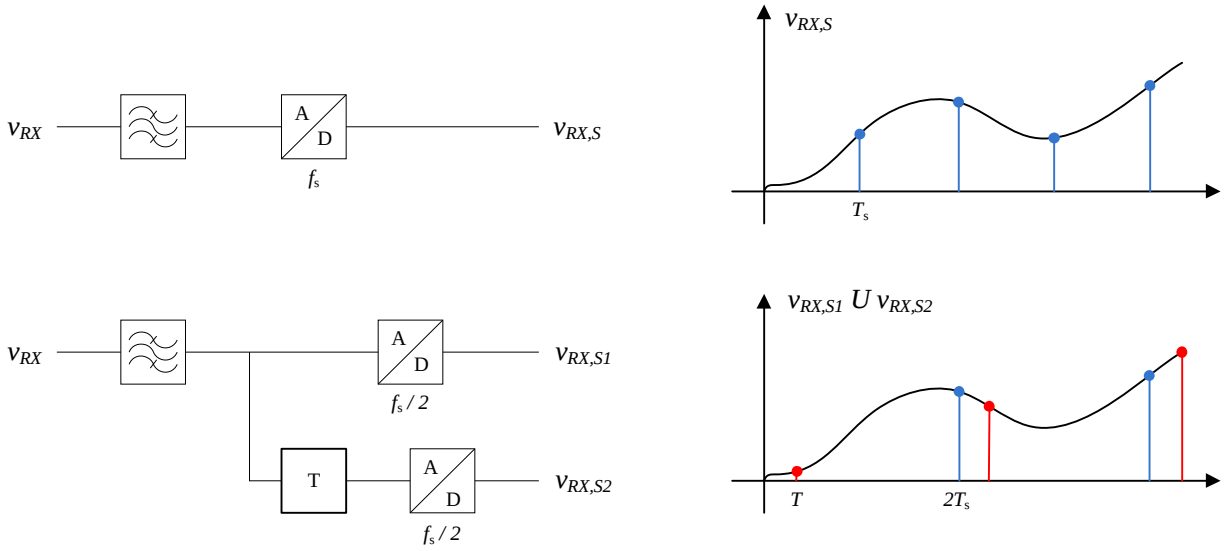

Fig. 2.7: First order and second order sampling.

Using these methods it is possible to make a hardware implementation avoiding more complex circuit elements that can introduce signal distortions.

2.3 Transmitter

As a part of the transceiver explained in the previous section, the transmitter is a part of other development as is explained in [11]. This transmitter is characterized by using a Gaussian baseband impulse and B-QPSK modulation.

The Gaussian impulse shape of the baseband signal *g(t)* in the time domain can be expressed as

$$g(t) = Ke^{\ln(g_t)\left(\frac{t}{\tau}\right)^2} = Ke^{-\pi(at)^2}, \tag{2.6}$$

where *K* denotes the peak amplitude for the time domain and τ represent the interval until *g(t*) is decayed to the value *Kg(t*). The frequency domain representation of the baseband impulse can be formulated as

$$G(f) = K\tau\sqrt{\frac{-\pi}{\ln(g_t)}}e^{\frac{(\pi\tau f)^2}{ln(g_t)}} \tag{2.7}$$

The constant $K\tau\sqrt{\frac{-\pi}{\ln(g_t)}}$ represents the peak amplitude for the frequency domain and the parameter τ is denoted as

$$\tau = \frac{2\sqrt{ln(g_t)(G_f)}}{B\pi}, \tag{2.8}$$

which is depending on G_f = -10 dB = 0.316, g_t which has been chosen to be 10% of the maximum amplitude, and the two-sided bandwidth $B = 2f_c$ = 500 MHz. Using the aforementioned data, the time parameter has the following value τ = 1.07 ns. An example of both *g(t)* and *G(f)* using a 0.5 peak amplitude and the aforementioned characteristics is given in the figure 2.8. As can be seen, the signal in the frequency domain preserves a Gaussian shape too.

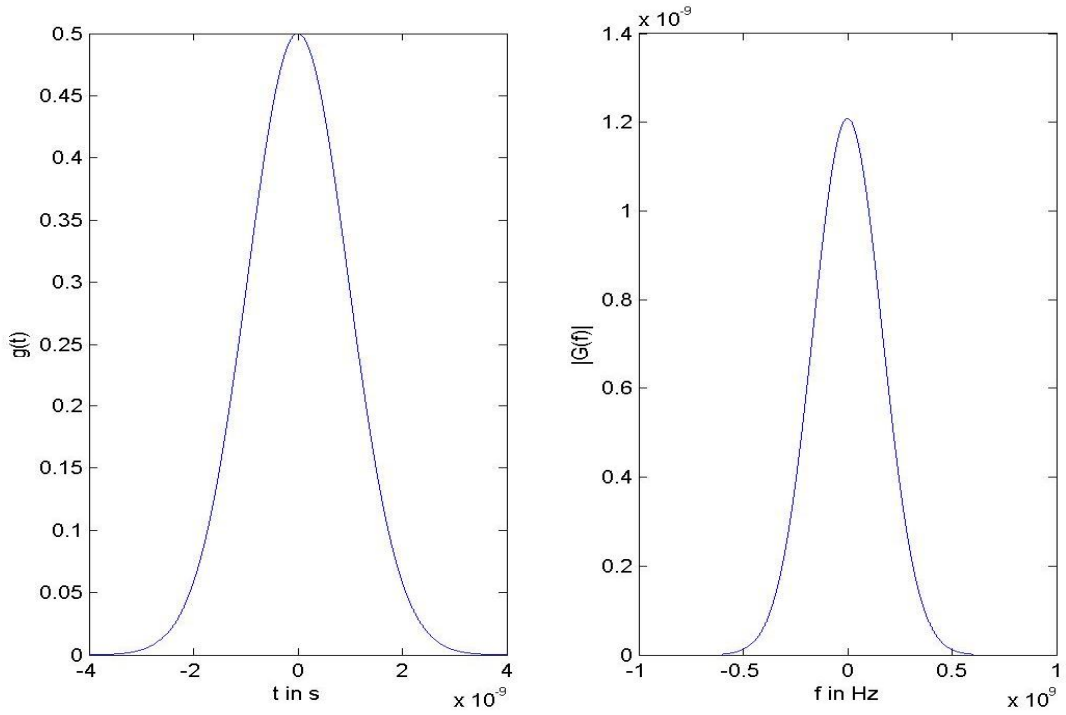

Fig. 2.8: Time and frequency domain representation of the baseband impulse.

Once the baseband impulse, the next step is to modulate the carrier signal using it. The carrier signal centred at the f_0 frequency can be formulated as

$$m(t) = \sqrt{2}\cos(2\pi f_0 t), \tag{2.9}$$

which modulated by the baseband impulse yields

$$g_m(t) = K\sqrt{2}cos(2\pi f_0 t)e^{ln(g_t)\left(\frac{t}{\tau}\right)^2}. \tag{2.10}$$

At the beginning of this section, it was only a Gaussian baseband impulse centred at the coordinate origin in the frequency domain but now, the impulse is moved to the f_0 frequency because of the modulation. This situation yields from the comparison of the figures 2.8 and 2.9. If the frequency domain is observed, the signal has moved from 0 Hz to f_0 frequency that in this case is 2 GHz. Moreover, each baseband impulse represents a symbol transmitted and it is replicated in the time domain with a T_{sym} period.

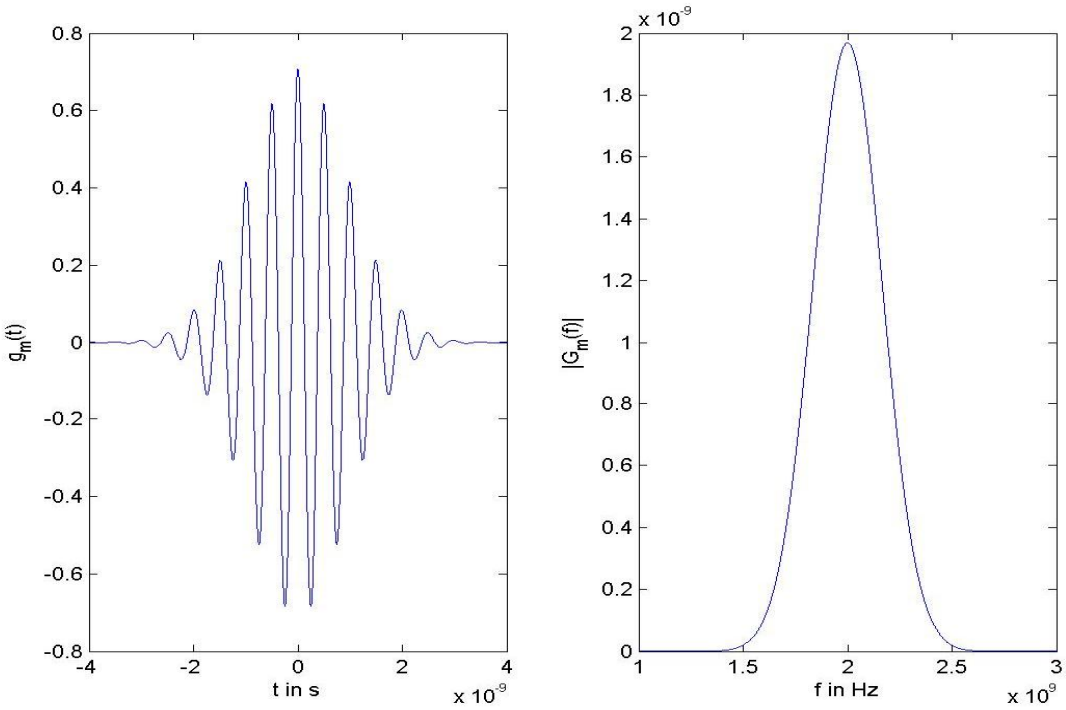

Fig. 2.9: Time and frequency domain representation of the modulated signal.

This prototype also uses B-QPSK modulation in order to be able to transmit different kind of symbols yielding

$$g_{\mathrm{m,PSK}}(t) = K\sqrt{2}cos(2\pi f_0 t + \theta_{\mathrm{PSK}} + \varphi)e^{ln(g_t)\left(\frac{t}{\tau}\right)^2} \tag{2.11}$$

where θ_{PSK} denote the symbol to transmit and φ stands for a phase modulation constant. As mentioned earlier, this system uses binary and quadrature that uses 2 and 4 symbols respectively. In the software implementation is contemplate the use of 8 or 16 symbols for the transmission but this is is unworkable in practice because of the lack of robustness to errors. The figure 2.10 shows an example of the constelation of a BPSK and QPSK modulation and after it, the table 2.1 indicates the different values of θ_{PSK} in order to get the different symbols.

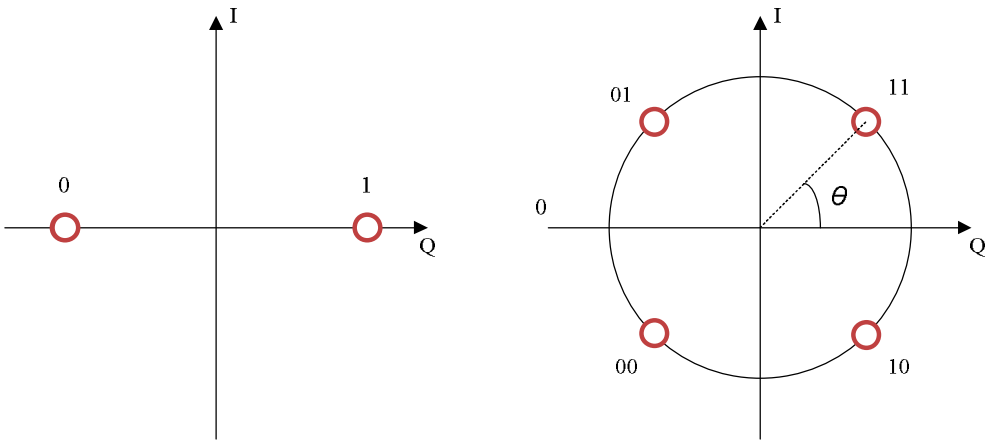

Fig. 2.10: BPSK and QPSK constelation modulation

Table 2.1: Values of θ_{PSK} to transmitt different symbols

	θ_{PSK}
BPSK	0º - 180º
QPSK	45º - 135º - 225º - 315º
8-PSK	0º - 45 - 90º - 135º - 180º - 225º - 270º - 315º
16-PSK	0º - 22,5º - 45º - 67,5 - 90º - 112,5º - 135º - 157,5º - 180º - 202,5º - 225º - 247º - 270º - 292º - 315º - 337,5º

Using the aforementioned considerations a possible transmitted signal is depicted in the figure 2.11.

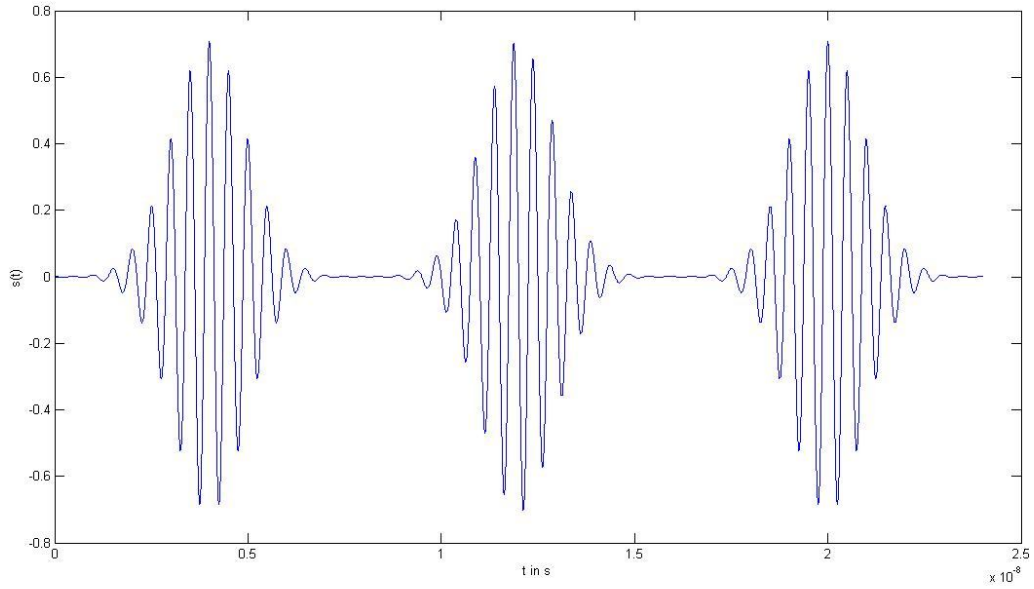

Fig. 2.11: Possible transmitted signal.

The FCC allows transmitting with UWB systems only within some spectral limits [2]. Note that the FCC has only specified a spectral mask and has not restricted users to any particular modulation scheme. For the characteristics of our prototype, the appropriate limit is an “Indoor Communication Systems” mask shown in the figure 2.12.

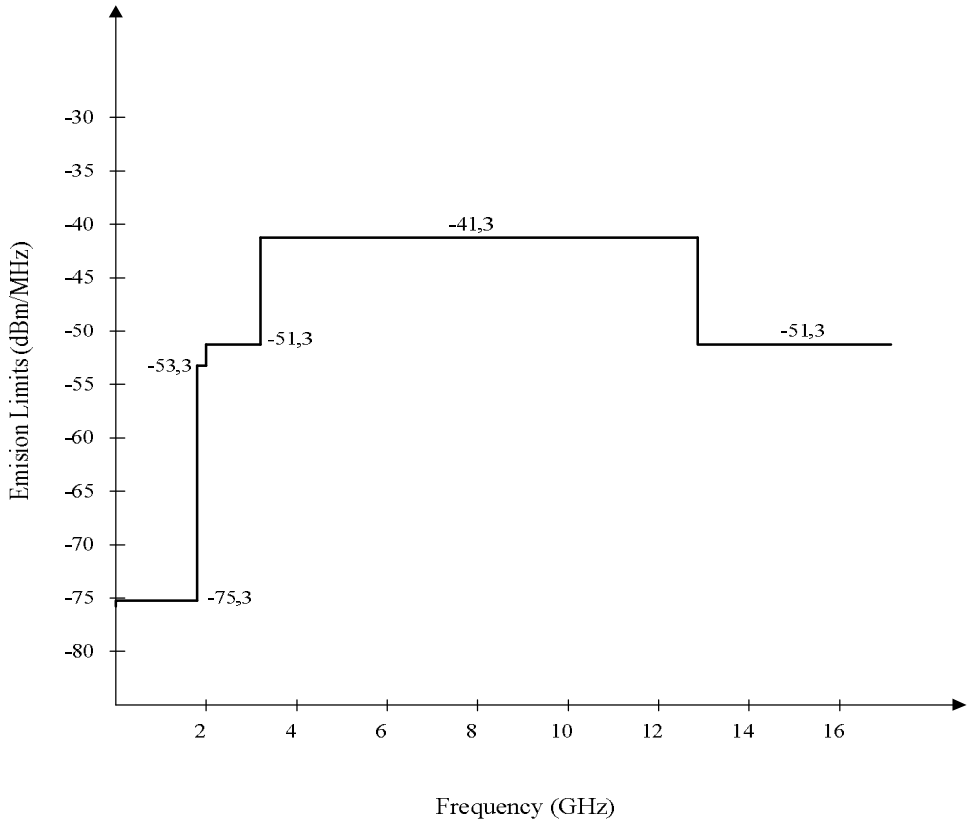

Fig. 2.12: Spectral Mask for Indoor UWB Communications Systems.

As can be seen in the figure 2.12, the transmit power limit of this application is -41.3 dBm/MHz. In order to calculate the power spectral density, the whole sequence of impulses has to be considered [2].

$$PSD(f) = \frac{\sigma_b^2}{T_f}\frac{|G_m(f)|^2}{Z_c} + \frac{\mu_b^2}{T_f^2}\sum_{m=-\infty}^{+\infty}\frac{|G_m(f)|^2}{Z_c}\delta\left(f - \frac{m}{T_f}\right) \quad (2.12)$$

(12) is given for an equidistant sequence of impulses b spaced by T_f. Z_c represents the characteristic impedance, whereas μ_b and σ_b^2 denote the expected value and the variance of the transmitted symbols, respectively. In order to calculate the maximum value of the PSD, it must be fixed at the centre frequency f_0, so in an approximation, for a 1 MHz signal, the PSD would satisfy

$$\int_{f=f_0-0.5\,MHz}^{f_0+0.5\,MHz} PSD(f)df \leq P_{max} \quad (2.13)$$

The maximum power P_{max} = 1mW = $10^{\left(\frac{-41,3}{10}\right)}$ and the previously specified parameters lead to a peak amplitude

$$K = \sqrt{\frac{-2ln(g_t)P_{max}Z_c}{\pi\tau^2\left(\frac{\sigma_b^2}{T_f}1MHz + \frac{\mu_b^2}{T_f^2}\right)}}, \quad (2.14)$$

3. RECEIVER DESIGN

In this section, a Software Defined Subsampling UWB B-/QPSK receiver is presented. The explanation is carried out from a theoretical point of view taking into account some hardware requirements and the concepts shown in previous sections.

A description of the entire receiver is summarized as a block diagram in the figure 3.1. As can be seen, the system receives the signal $g_{\mathrm{RX}}(t)$. This signal consists of the transmitted signal $g_{\mathrm{m,PSK}}$ modified by the effects of the channel and noise. The influence of this factors results in

$$g_{\mathrm{RX}}(t) = g_{\mathrm{m,PSK}}(t) * h(t) + n(t), \tag{3.1}$$

where $h(t)$ denotes the channel impulse response and $n(t)$ is modelled as additive white Gaussian noise (AWGN). The channel response is calculated using some coefficients of a finite impulse response filter.

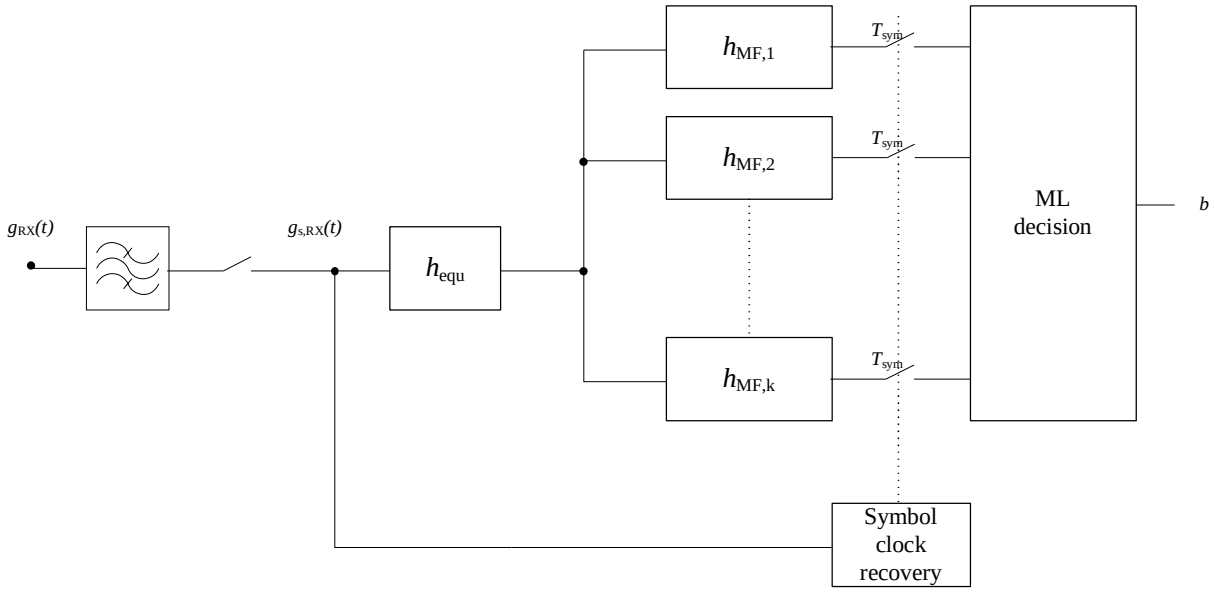

Fig. 3.1: Block diagram of the receiver prototype.

The next step in the diagram is to sample the received analog signal with a period T_s yielding

$$g_{s,RX}(t) = g_{RX}(t) \sum_{n=-\infty}^{\infty} \delta(t - nT_s) \tag{4.2}$$

which has a spectrum defined by

$$G_{s,RX}(f) = G_{RX}(f) \frac{1}{T_s} \sum_{k=-\infty}^{\infty} \delta\left(f - \frac{k}{T_s}\right) \tag{4.3}$$

At this moment, the sampled signal is located at RF but it is necessary to down convert it to the baseband. The downconversion usually requires additional mixers stages but as it was explained previously, it causes signal distortions due to their non-ideal transfer characteristic. In order to avoid the distortions and to make the hardware design simpler, it is possible to benefit of the properties of a discrete signal as it was done in the transmitter stage. Proceeding in the same way done in the transmitter, the signal (3.3) is periodic and it has replicas in the different Nyquist zones. Using a simple lowpass filter provides the desired replica in the first Nyquist zone. This situation is depicted in the figure 4.2.

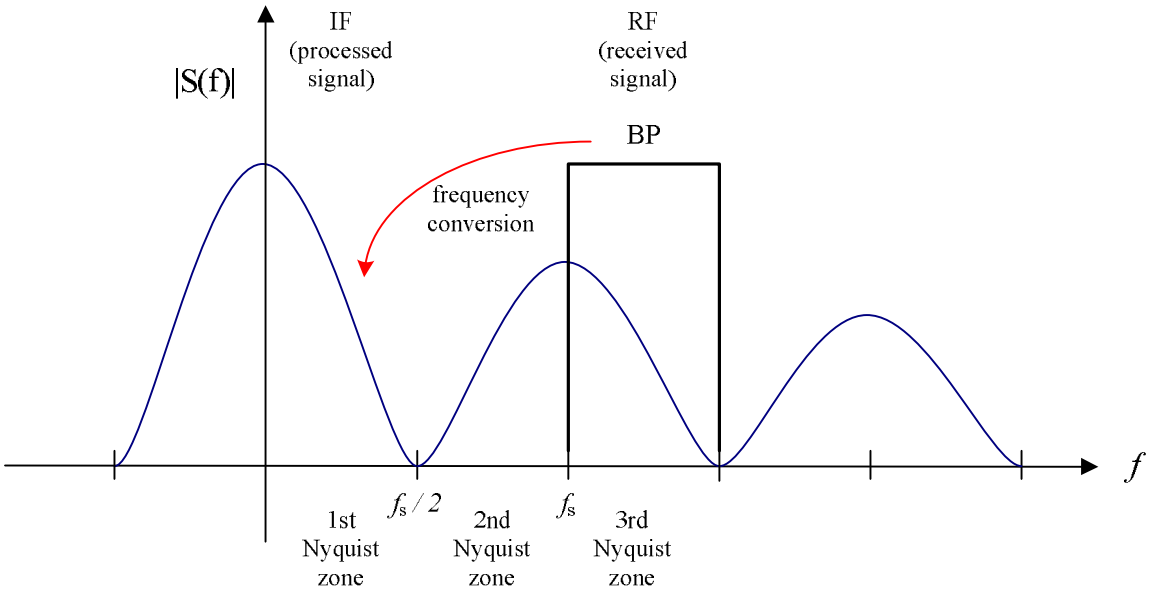

Fig. 3.2: Spectrum of a time discrete signal.

The rest of the process consists of a matched filter structure. The function of the matched filter is to detect the presence of a known signal inside an unknown noisy signal received. This system is the optimal linear filter for maximizing the signal to noise ratio (SNR) defined by

$$\text{SNR} = \frac{\text{signal power}}{\text{noise power}} \tag{3.4}$$

in the presence of AWGN. This is a very good measurement of the quality of a demodulator. The output of matched filter is obtained by correlating a known signal, or template, with an unknown signal to detect the presence of the template in the unknown signal. This is equivalent to

convolving the unknown signal with a conjugated time-reversed version of the template (cross-correlation) [12] formulated as

$$g_{\mathrm{MF}}(t) = g(t) * h^*_{\mathrm{MF}}(-t) = \int g(t) * h^*_{\mathrm{MF}}(\tau - t)\, dt \tag{3.5}$$

that in discrete time is given by

$$g_{\mathrm{MF}}[n] = \sum_{k=-\infty}^{\infty} g[k] * h^*_{\mathrm{MF}}[n-k]. \tag{3.6}$$

The result of the cross-correlation measures the similarity between the template and the received signal. This operation is done with all the different templates of each symbol and the symbol chosen is the one with highest correlation.

In this case, the sampled signal (3.2) is convolved with the equalizer impulse response $h_{\mathrm{equ}}(t)$ which is the inverse of the channel impulse response. This allows removing the channel effects and provides the original signal with noise added $g_{\mathrm{n}}(t)$. (As we said previously, the estimation of the channel response is not implemented in the software solution). The original signal with noise $g_{\mathrm{n}}(t)$ is correlated with the template of each symbol $h_{\mathrm{MF,k}}(t)$ which would be the same as the modulated impulse $g_{\mathrm{m,PSK}}(t)$ given in the equation (2.10). The result is sampled with T_{sym} period given by a symbol clock recovery and finally, the symbol chosen for each sample is the one with highest correlation.

The matched filter process is graphically explained in the next figures. In the figure 3.3 can be seen a possible BPSK transmitted signal sending a 0100 sequence.

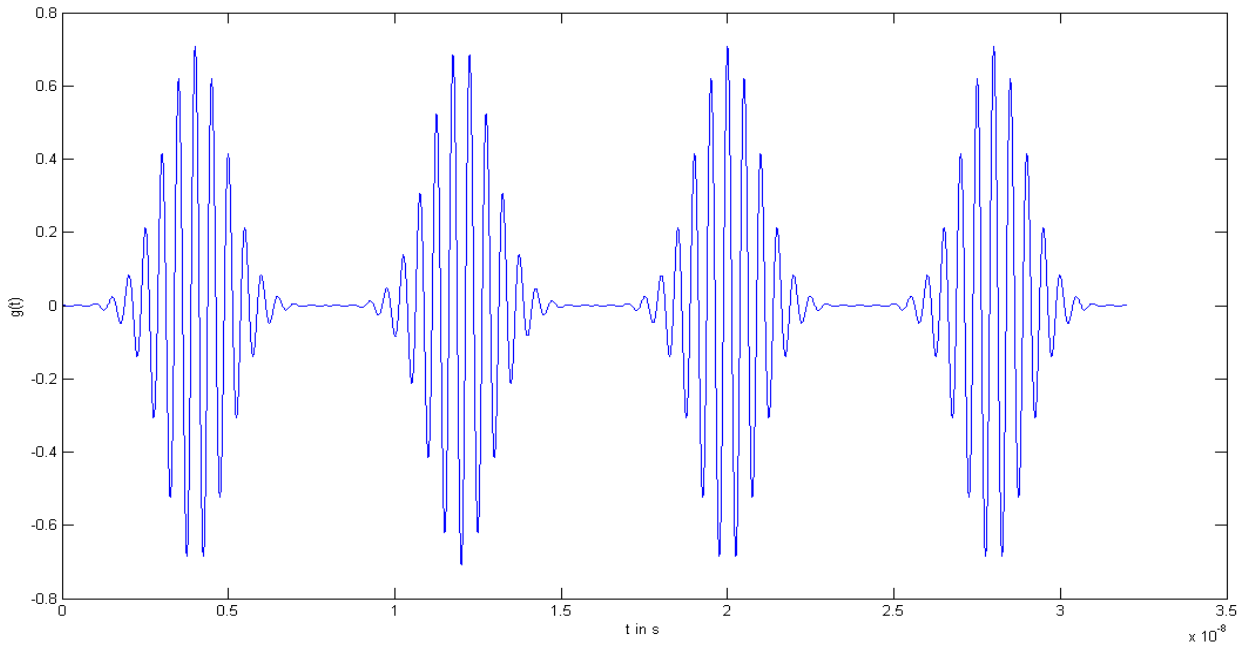

Fig. 3.3: BPSK transmitted signal. Sequence transmitted 0100.

The receiver signal after the equalizer is depicted in the figure 3.4. As can be seen, the signal has a very low SNR and the channel efects are completely removed. Normally in a real case, it is not possible to completely remove the channel and the signal received has a lot of distoritions, but this is an ideal case.

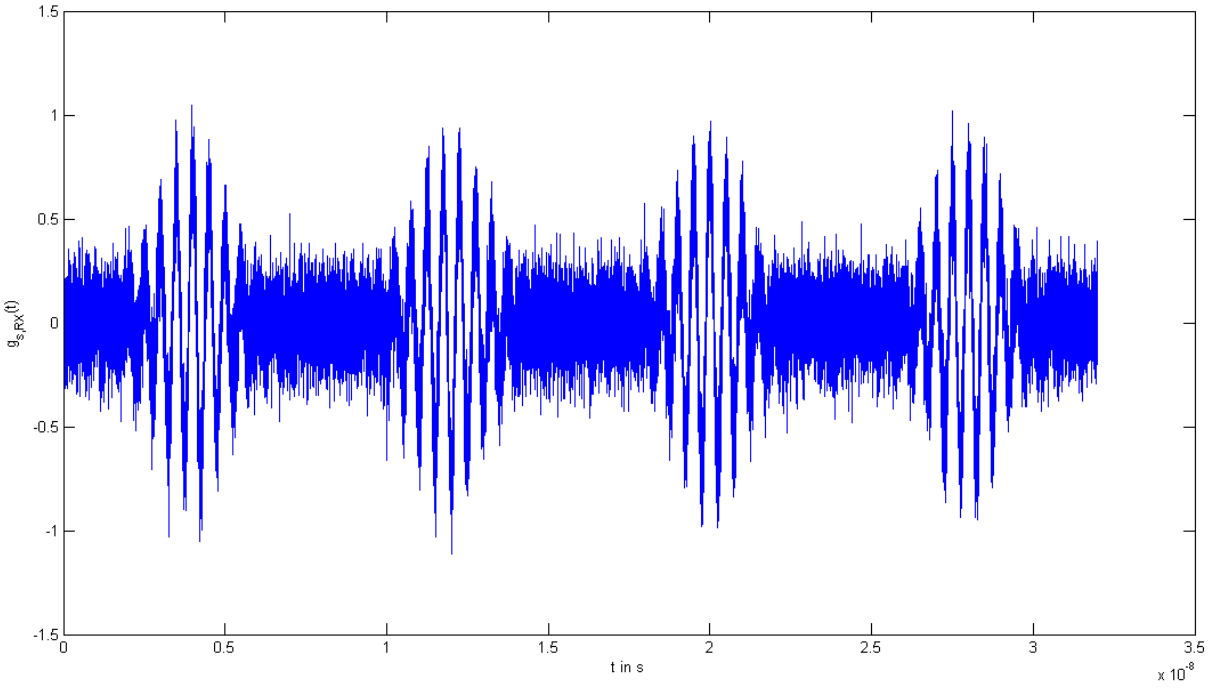

Fig. 3.4: Received signal with AWGN.

The signal pass through the matched filter and the result is shown in the figure 3.5. In this case, the chosen modulation was BPSK and that means that there will be two signals after the matched filter. One of these signals will be the cross-correlation with the '0' template and the other signal will be the cross-correlation with the '1' template shown in the figure 3.5 top and figure 3.5 bottom respectively. The following step is to decide which the transmitted symbols are. In order to achieve this, it is necessary to sample the outputs with a T_{sym} period. Furthermore it is essential that the receiver is synchronized with the transmitter. If this requirement is not achieved, the received bits can be totally wrong. Finally, the symbol chosen for each sample is the one with highest correlation.

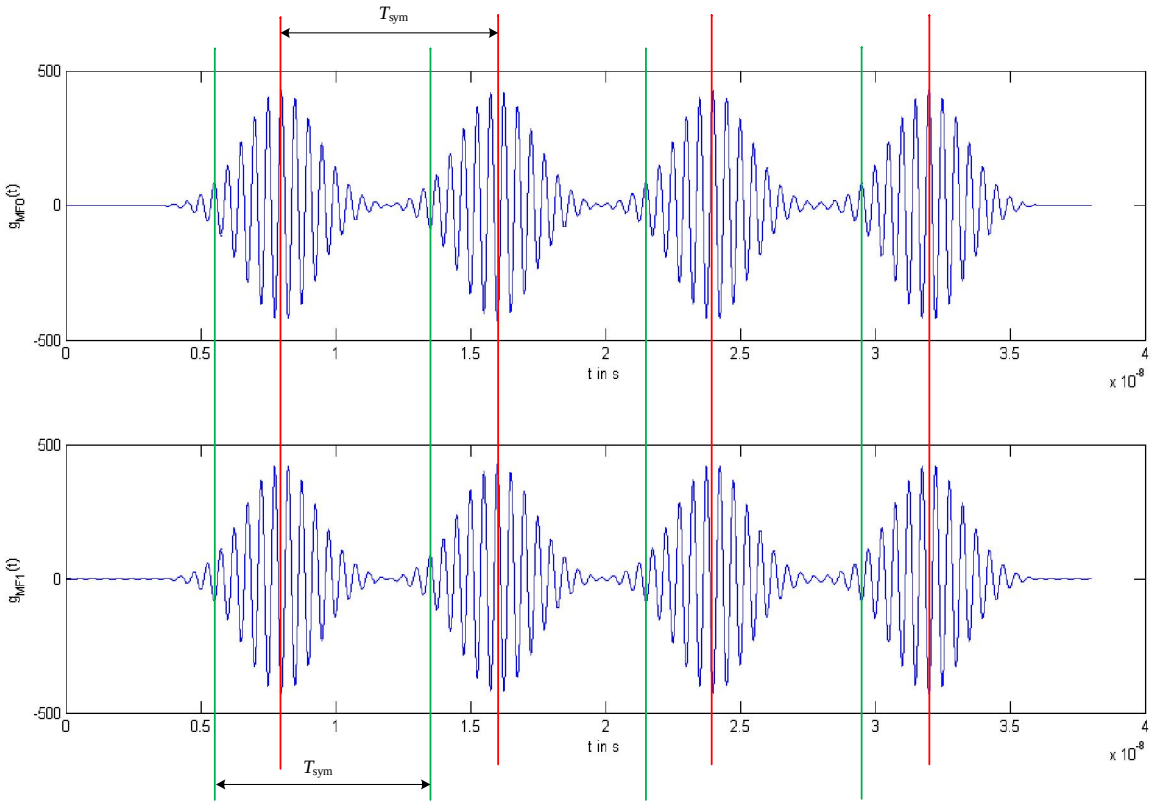

Fig. 3.5: Matched filter output

4. SOFTWARE RECEIVER SOLUTION

A software receiver solution is presented in this section. As it was commented previously, the principal issues of the software solution are to make possible the communication between Matlab and C, and to get a suitable solution for the receiver.

The solution for the communication between Matlab and C is given using Mex-files. There is a function called "*qpsk_receiver()*" written in C, where all the reception process takes place. This function also use other complementary functions needed for carry out some operations in the reception process. All these concepts are explained in the following sections.

4.1 Mex-files

The first problem that is going to be argued is the communication between Matlab and C. This problem was solved using Mex-files. Mex-file is an external interface of Matlab that allows calling your custom C or Fortran routines directly from Matlab as if they were Matlab built-in functions [13]. In this way, this system provides the ability of transfer data from Matlab to C or other functions avoiding saving the data as Mat-files and the consequent time needed in order to make this operation. Thus, this model allows to decrease the computational time of some subroutines e.g. boucles that in Matlab requires more computational time.

Applying this information for this problem, Mex-files are a very efficient solution. In the Matlab solution of the transceiver, all the data is transferred to all the different functions saving it as Mat-file. This requires more computational time that transfer the data as a subroutine parameter using the memory of the pc. Thus, this technique allows to this prototype to save computational time.

The structure of a Mat-file is the following [14]:

```
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]) {

   //All code and internal function calls go in here!

   return;

}
```

This structure must be the same in all the Mex-files. In order to call this function from Matlab, it should be used the name of the file saved with ".c" extension. The meaning of the parameters is the following:

- nlhs: Number of "left hand side" arguments. "Left hand side" is equivalent to the "left hand side" in Matlab. They are parameters that the subroutine resturns.
- nrhs: Number of "right hand side" arguments. "Right hand side" is equivalent to the "right hand side" in Matlab too. They are parameters that the subroutine receives.
- plhs: This parameters is an array of pointers to mxArrays that contains the output arguments. mxArray is a Mex-file vector.
- prhs: This parameters is an array of pointers to mxArrays that contains the input arguments.

All the Mex-files must include something more elements in his code. Those elements are the following:

- "#include mex.h" (C/C++ MEX-files only).
- mexFunction gateway in C/C++ (or SUBROUTINE MEXFUNCTION in Fortran).
- The mxArray.
- API functions.

The subroutine must be compiled. In order to achieve this, the following sentence must be typed at the Matlab prompt: mex nameofthefile.c.

Mex-files have their own structures and functions in order to access, create, destroy... variables. This can be founded in [15]. Some of the functions used in this solution are the following:

- *mexErrMsgTxt(errormsg)*. This function displays an error message and return to MATLAB prompt. "*errormsg*" is a string containing the error message to be displayed.
- *double *mxGetPr(const mxArray *pm)*. This function returns the address of the first element of the real data in the mxArray that "*pm*" points to.
- *mxArray *mxCreateDoubleMatrix(mwSize m, mwSize n, mxComplexity ComplexFlag)*. This function returns a pointer to the created mxArray. The dimensions of the mxArray are specified by the variables m and n. "*m*" denotes the desired number of rows, "*n*" the desired number of columns and "*ComplexFlag"* the kind of numbers used for the array.

The following code is a simple example of a mex-function and it shows how it works.

```
#include "mex.h"
void vecProd();
void mexFunction();

void comblin(const double* a, const double* b, double* c) {
        c[0] = 5*a[0]-2*b[0];
        c[1] = 5*a[1]-2*b[1];
        c[2] = 5*a[2]-2*b[2];
}
```

```
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]) {
        double *a, *b; // inputs
        double *c; // outputs

// Check for proper number of arguments
        if(nrhs != 2) {
                mexErrMsgTxt("Error in number of inputs.");
        } else if (nlhs != 1) {
                mexErrMsgTxt("Error in number of outputs");
        }

// Create matrix for the return argument
        plhs[0] = mxCreateDoubleMatrix(3, 1, mxREAL);

// Assign pointers to each input and output
        a = mxGetPr(prhs[0]);
        b = mxGetPr(prhs[1]);
        c = mxGetPr(plhs[0]);

// Call the function written in C
        vecProd (a,b,c);
}
```

This example provides a linear combinatory of two vectors with three elements. The code would be saved in a file with the name "product.c". This function will be called by Matlab in the form "c=product(a,b)", where "*c*" would be the output and "*b*" and "*c*" would be the inputs. When the function is called, the execution of the code starts at "*mexFunction()*".

Once inside the "*mexFunction()*", the first step is to check the number of outputs and inputs. In this case the number of inputs and outputs needed are two and one respectively. If this requirement is not satisfied, the function would display an error message and return to Matlab prompt. The inputs would be two vectors of three elements and the output would be a vector of three elements where the results would be stored.

The next step in this routine would be to create a vector of three elements in order to store the results that Matlab receives. This vector would be creating in the first position of the "*plhs*" array using the "*mxCreateDoubleMatrix()*" method. After this, the variables used in the "*mexFunction()*" must be related with the variables stored in the "*plhs*" and "*prhs*". This is achieved using "*mxGetPr()*" function. In this way, the variables "*a*" and "*b*" are assigned to the input pointer and the variable "*c*" is assigned to the output pointer.

Finally, the method that makes the linear combination "*comblin()*" is called inside the "*mexFunction()*" transferring to him as parameters the input and output variables. Inside this function, the pointer that contains the solution is modified in order to store the result inside it.

In the receiver problem, the function of the Mex-file is to transfer the data and call the receiver programmed in C language. The procedure employed is almost the same that has been used in the aforementioned example. The function returns two vectors that contain the bits and symbols demodulated vectors. It recibes as parameters the rest of the parameters that the *qpsk_receiver()* function needs. A flowchart of the proposed solution is given in the figure 4.1.

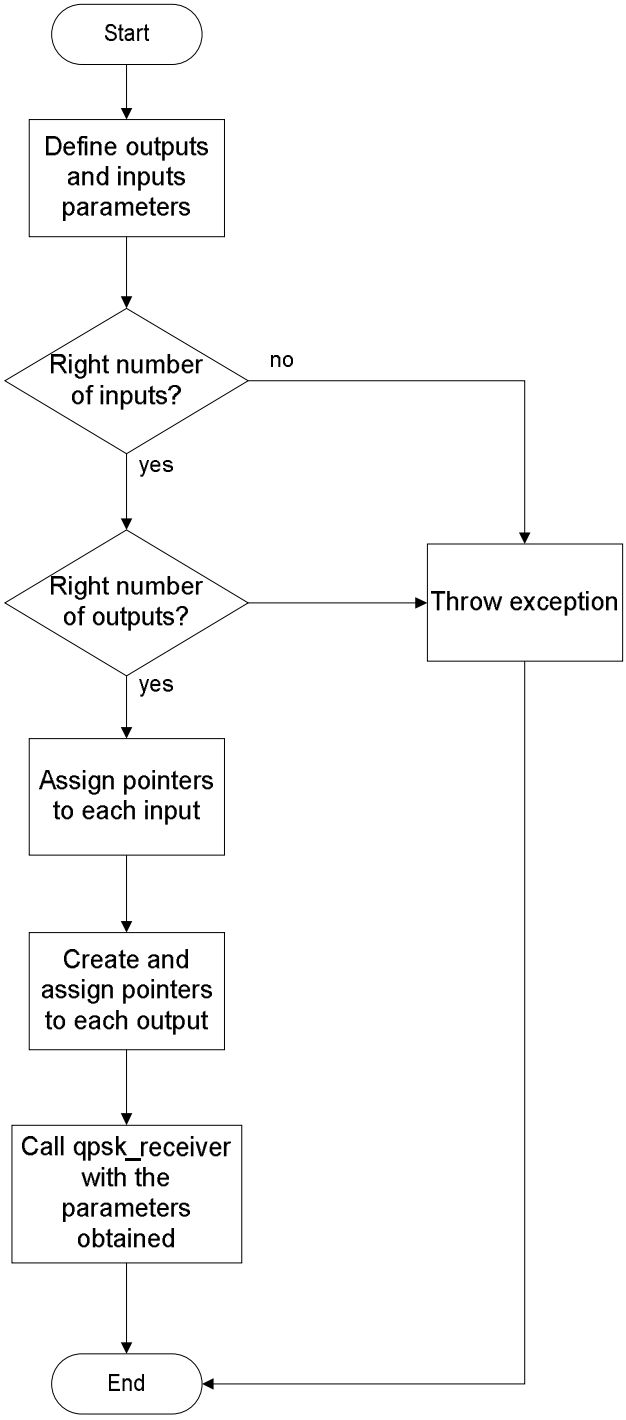

Fig. 4.1: Flowchart of the "*mexfunction()*".

As in the example given, the "*mexfunction()*" has "translator" role. The routine that carries out the reception and demodulation of the data is "*qpsk_receiver()*".

4.2 QPSK Receiver

In this section, the "*qpsk_receiver()*" function is described. Inside this function takes place all the reception process and it is written in C language. The header of this routine is determined by;

void qpsk_receiver(double f_s, int n_samples_symbol, double f_c, double b, double z_c, double p_max, double g_t, double g_f, double tau, double sigma_b, double mu_b, double K, double lambda_mod, int m_psk, double rec_sigma_noise, double snr, double sigma_jit, double channel_d, double n_sym, double rec_n_sym, int scenario, double tx_bit, double* tx_sym, double* rx_in, double* h_bp, double* rx_bit_total, double* rx_sym_total)*

f_s = Sampling frequency.
n_samples_symbol = Number of samples per symbol.
f_sym= f_s/n_samples_symbol = Symbol frequency.
f_c = Carrier frequency.
b = 10 dB bandwidth of Gaussian baseband impulse.
z_c = Characteristic impedance, usually 50 Ohms.
p_max = Maximum allowed transmit power in 1 MHz.
g_t = Minimum amplitude of impulse.
g_f = Attenuation at corner frequency.
tau = time constant.
sigma_b = Standard deviation of transmitted pulse train.
mu_b = Mean value of transmitted pulse train.
K = Scale factor.
*lambda*_mod = Phase modulation constant.
m_psk = Number of different symbols that it is possible to transmit.
n_sym = Number of symbols to be generated.
rec_n_sym = number of symbols for best timing.
scenario = Kind of scenario.
tx_bit = Bits transmitted.
tx_sym = Symbols transmitted.
rx_in = Noise.
rx_bit_totat = Bits received.
rx_sym_totat = Symbols received.

This function can be structured in several distinct parts and this parts can be used in order to descrive in a general way the "*qpsk_receiver()*". The routine starts creating baseband and modulates templates exactly in the same way that the transmitter does. Once created the modulated templates, they are used to create the matched filters templates. Using the matched filters templates, the synchronization is carried out and finally the decision of the symbols and

bits received takes place. This process, as we aforementioned summarizes the receiver routine as is shown in the figure 4.2.

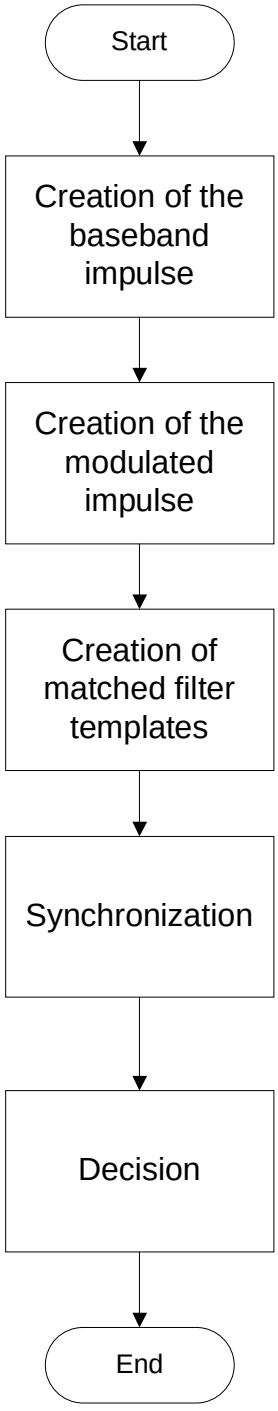

Fig. 4.2: General diagram of the receiver solution.

The process of the generation of the baseband impulse uses the same funds than the transmitter. The baseband impulse has the same shape given by the equation (2.6)

$$g(t) = Ke^{\ln(g_t)\left(\frac{t}{\tau}\right)^2} = Ke^{-\pi(at)^2}, \tag{4.1}$$

The samples of the baseband impulse are generated using this equation with a sampling frequency given by $f_s = 1/t$. The length of the pulse will depend on the amplitude of the mentioned impulse. The maximum amplitude will be reached at the value given by Kg_t, thus the half of the impulse will be reached when the amplitude obtained for a certain sample is higher than Kg_t. At this moment, the second half of the impulse is obtained copying the symmetrical part. Finally, the rebounds in both sides of the impulse are deleted using a tukey window of the same length.

Continuing with the next block of the figure 4.2, is found the process of the generation of the modulates impulses. As the baseband impulse, the modulated impulses are generated using the same shape as the transmitter given by the equation (2.11)

$$g_{m,PSK}(t) = K\sqrt{2}cos(2\pi f_0 t + \theta_{PSK} + \varphi)e^{ln(g_t)\left(\frac{t}{\tau}\right)^2} \tag{4.2}$$

The samples of all the different modulated templates are generated using this equation with the same sampling frequency f_s as the baseband impulse. The different templates are built by different values to θ_{PSK} given by the table 2.1. The number of values would depend on the number of symbols that can be transmitted. The length of the modulated templates vectors would be the number of samples that are transmitted by each symbol.

Once finished the generation of the impulses templates, the code start working with the bit stream sequence received. The software simulator system used to work jointly with the receiver sends the bit stream to the receiver without considering the noise channel effects. For this reason, the routine adds to the received signal AWGN using the function "*add_awgn()*".

The next step in the diagram of the figure 4.2 is to generate the matched filter templates. As it was explained in the previous sections, the matched filter templates are obtained using the cross-correlation given by the equation (3.5)

$$g_{MF}(t) = g(t) * h^*_{MF}(-t) = \int g(t) * h^*_{MF}(\tau - t)\, dt. \tag{4.3}$$

Each matched filter template is obtained through the convolution of the received signal and the corresponding inverted modulated impulse. In order to carry out this operation the functions "*conv()*" and "*fliplr()*".

After obtaining the matched filter templates, the synchronization takes place. This process is mixed with the decision process and depending on its success, the receiver design will be a bad or good solution. The synchronization is done using a small sequence at the beginning of the signal. The number of the symbols used for this task is received by the "*qpsk_receiver()*" as a parameter by a variable called "*rec_n_sym*". The procedure consists of taking one sample by

symbol for each template of the matched filter output with the f_{sym} frequency until the number of symbols indicated by "*rec_n_sym*" is reached. After obtaining the samples, the decision is done for each symbol using the matched filter criterion. As it was explained in the previous section, the symbol chosen for each sample is the one with highest correlation or matched filter output. Once all the first "*rec_n_sym*" symbols are decided, the symbol error rate is calculated comparing with the real symbol stream transmitted. All this process is done a certain number of times changing a time offset for each one. The different samples obtained for all the different time offset is depicted in the figure 4.3.

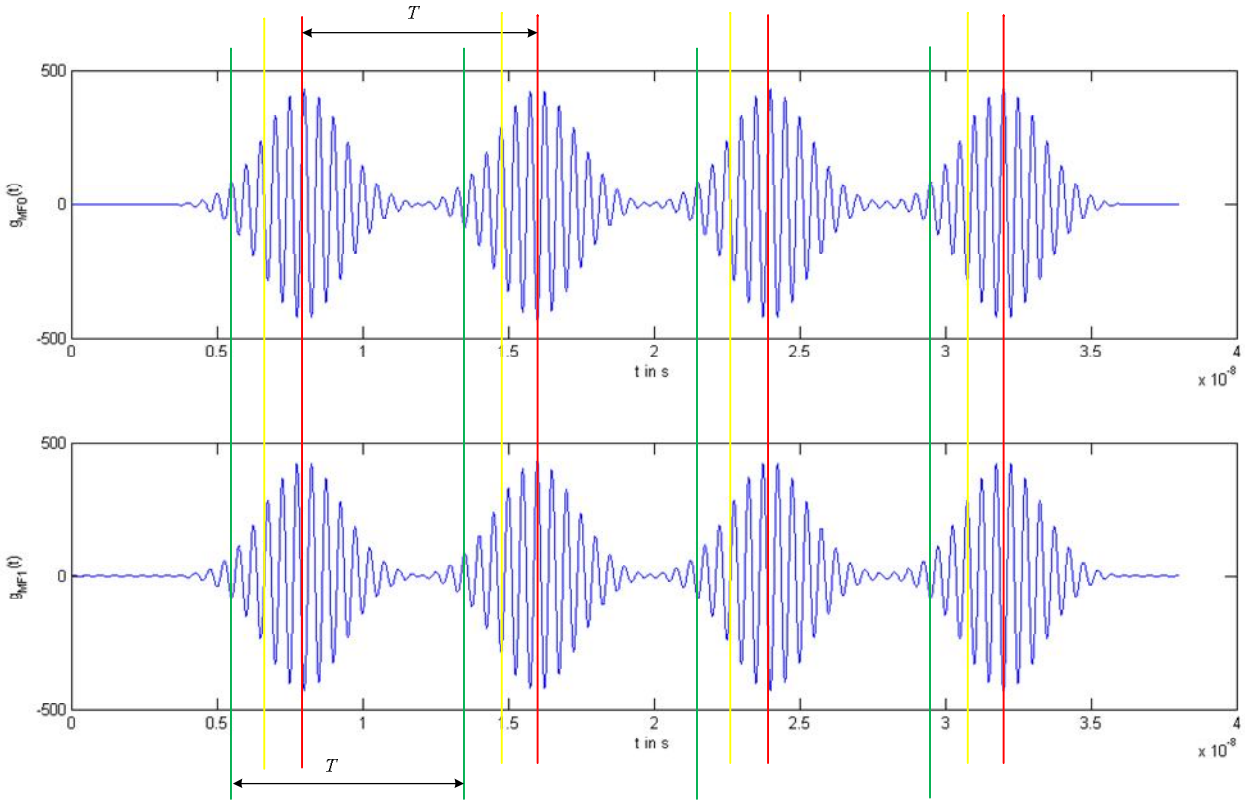

Fig. 4.3: Samples obtained with different time offset.

Finally all the different symbol error rates for each time offset are compared. The time offset with lowest symbol error rate is chosen and it will used for demodulating the entire symbol stream received. Then, the symbol stream is converted into bit stream.

The following figure shows a flowchart of the "*qpsk_recever()*" function.

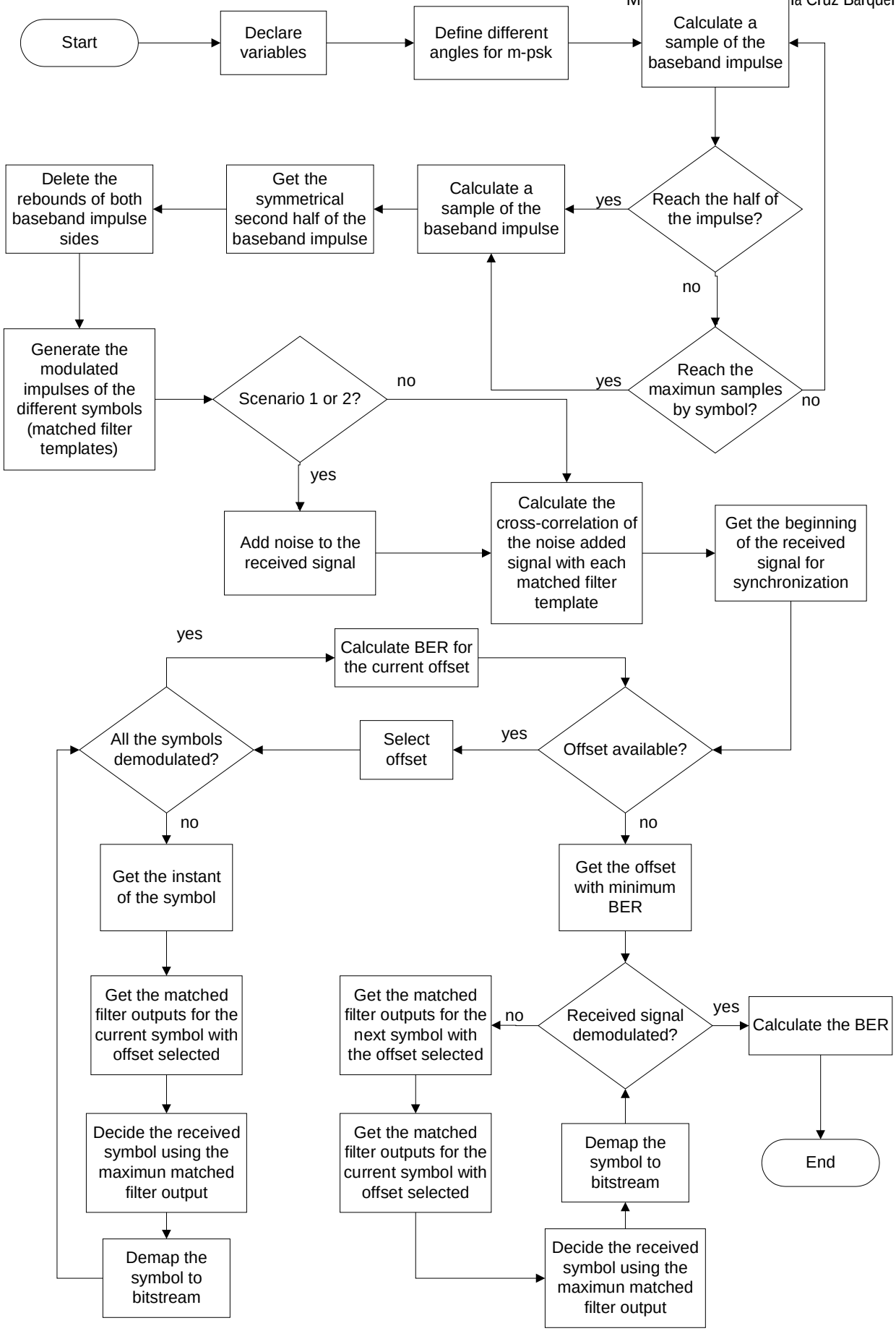

Fig. 4.4: Flowchart of the receiver solution.

4.3 Complementary Functions

In this part some complementary functions implemented for the "*qpsk_recever()*" function are presented. These functions appear of the need of some vector operations and signal processing that are used in the receiver.

- ***void add_awgn(double* rx_in, double snr, int long_rx, double* final_signal)***

This function allows to add AWGN to a signal. It receives as parameters the origin signal where the noise is going to be added "*rx_in*", the SNR required "*snr*", the length of the origin signal "*long_rx*" and the variable were the result is going to be stored "*final_signal*".

The flowchart of this function can be seen in the figure 4.5.

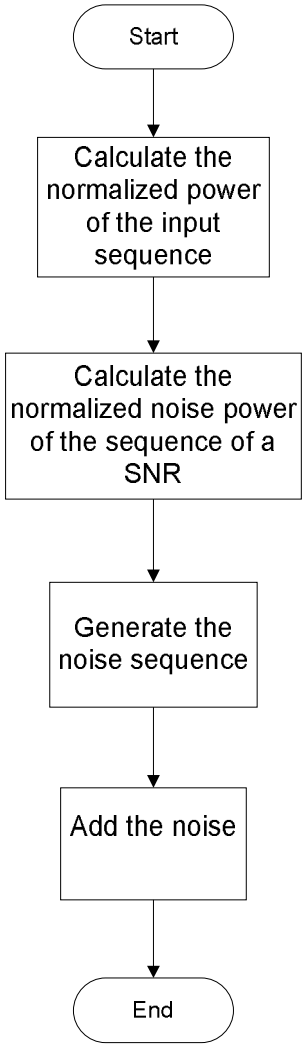

Fig. 4.5: Flowchart of the "*add_awgn()*" function.

The normalized power of the input sequence is calculated second the following equation

$$PNorm = \frac{1}{T\, Z_{\text{in}}\, T_{\text{s}}} \int |x(t)|^2\, dt, \tag{4.4}$$

where T denotes the interval of the signal where the power is evaluated, Z_{in} means the system impedance and T_{s} is the sampling interval. The normalized noise power of the sequence is obtained using

$$PNNorm = {PNorm}\Big/{10^{\frac{SNR}{10}}} \tag{4.5}$$

Finally, the noise sequence determined by

$$NS = randn \sqrt{PNNorm\, Z_{\text{in}}\, T_{\text{s}}} \tag{4.6}$$

is calculated and added to each sample of the original signal. "*randn*" denotes values drawn from a normal distribution with mean zero and standard deviation one.

- ***void tukeywin(int L, double r, double* window)***

This function calculates an L-point, Tukey window in a vector "*window*". Tukey windows are cosine-tapered windows. "*r*" is the ratio of taper to constant sections and is between 0 and 1. The equation used for calculating the coefficients of the window is

$$w(n) = \begin{cases} 1, & 0 \le |n| \le r\frac{N}{2} \\ \frac{1}{2}\left(1 + cos\left(\pi \frac{n - r\frac{N}{2}}{2(1-r)\frac{N}{2}}\right)\right), & r\frac{N}{2} \le |n| \le \frac{N}{2} \end{cases} \tag{4.7}$$

where $L = N + 1$ denotes de window length. This function is used by the "*qpsk_receiver()*" function to delete the rebounds in the baseband impulse.

- ***void conv(double* v, int n, double* u, int m, double* result)***

This function convolves the vectors "v" and "u" with the lengths "n" and "m" respectively. The result of this operation has a length of $n + m - 1$ and it is stored in the variable "*result*". The convolution of two signals is given by

$$w(t) = u(t) * v(t) = \int_{-\infty}^{\infty} u(\tau) v(\tau - t) d\tau \tag{4.8}$$

for the continuous time whereas for the discrete time yields

$$w(k) = \sum_j u(j) v(k - j) \tag{4.9}$$

The equation 4.9 is used to build this routine. In order to obtain the convolved signals, a sum over all the values of "*j*" must be done. It is easy to calculate it using loops and other features of the programming language whenever some limits to the values of "*j*" are imposed. The range of values that "j" will have for each iteration in the summation include the interval

$$j = [\max(0, k + 1 - n) : \min(k, m - 1)] \tag{4.10}$$

This routine is used in the "*qpsk_receiver()*" function to calculate the matched filter templates.

- ***void fliplr(double* vector, int n_v,double* invert)***

This function calculates the vector "*result*" of the same length of the vector "*vector*" but with the order of its elements reversed. The variable "*vector*" is the vector that is going to be flipped, "*m*" represents the length of "*vector*" and "*result*" is the variable where the flipped vector is going to be stored. This function is used in the "*qpsk_receiver()*" function to create the baseband impulse and the matched filter templates.

- ***double maxi(double *vector, int n_v)***

This function returns the largest element along the array "*vector*" of a length given by "*n_v*". It is useful in the "*qpsk_receiver()*" function to make the decision between different symbols using the matched filter templates.

- ***double randn(void)***

This function returns a pseudorandom, scalar value drawn from a normal distribution with mean 0 and standard deviation 1. It is used in the "*add_awgn()*" function helping to create the noise sequence.

5. RESULTS AND SIMULATIONS

In this paragraph the results of simulating the software program created are presented. The results are shown using different figures that are obtained representing the signals during the execution process.

The parameters used for carry out this simulation are a sampling frequency f_s = 2 GHz, 32 samples per symbol, a minimum amplitude for the impulse g_t = 10%, a carrier frequency f_c = 500 MHz, an SNR = 5 dB, a characteristic impedance Z_c = 50Ω, K = 0.1442, a time parameter τ = 2.0731 ns, μ_b = 0, σ_b = 1, a phase modulation constant φ= 0, a bandwidth of 500 MHz, G_f = -10 dB and a maximum allowed transmit power for 1 MHz of P_{max} =1 mW $10^{(-41,3/10)}$.

A QPSK random signal is generated using the aforementioned data. The constellation diagram of the four symbols that can be transmitted is represented in the figure 5.1.

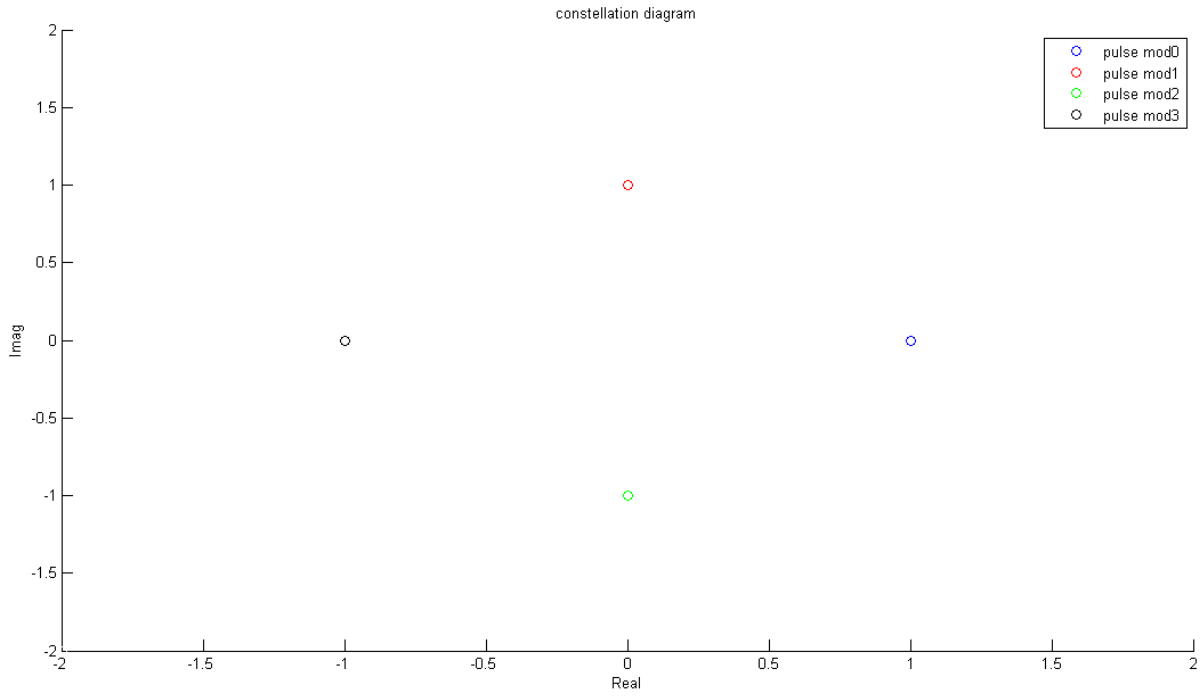

Fig. 5.1: Constellation diagram of QPSK signal.

The different modulated impulses shapes that represent each symbol can be observed and compared in the figure 5.2.

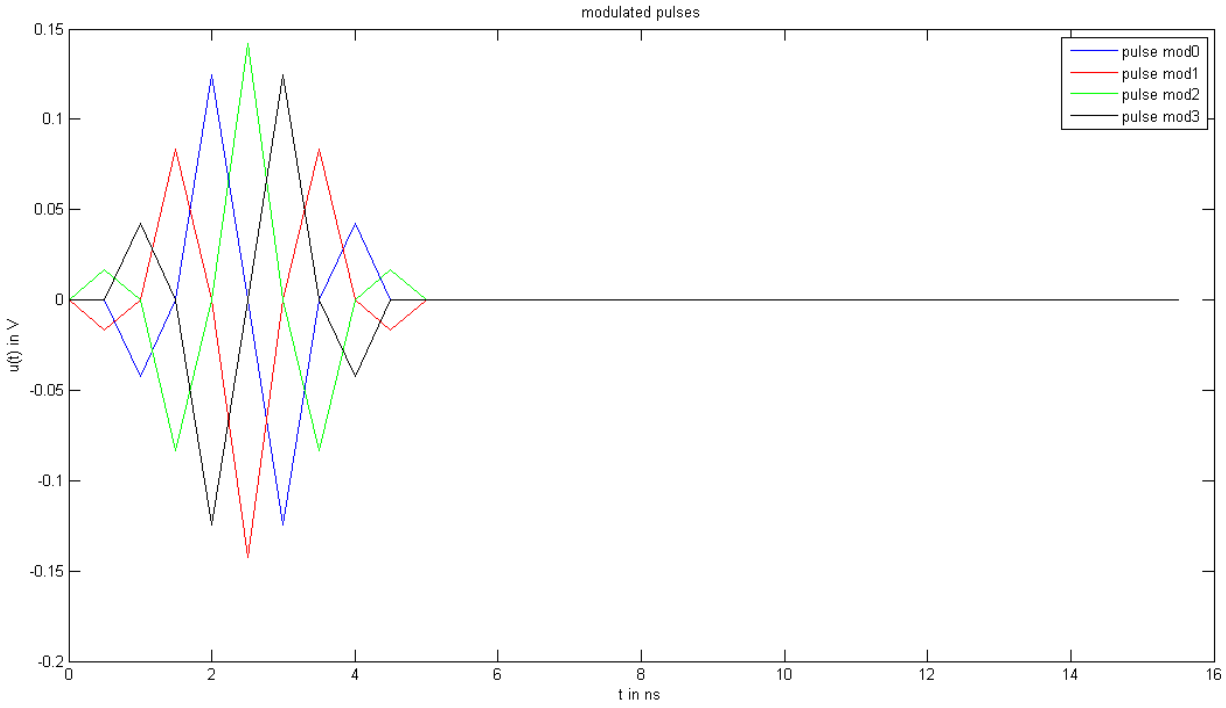

Fig. 5.2: Modulated impulse shapes.

Then, a random sequence of 50 symbols is generated. The digital signal is represented in the figure 5.3 as a bit stream signal. Each two bits would conform each symbol. The symbol stream signal is also represented in the figure 5.4.

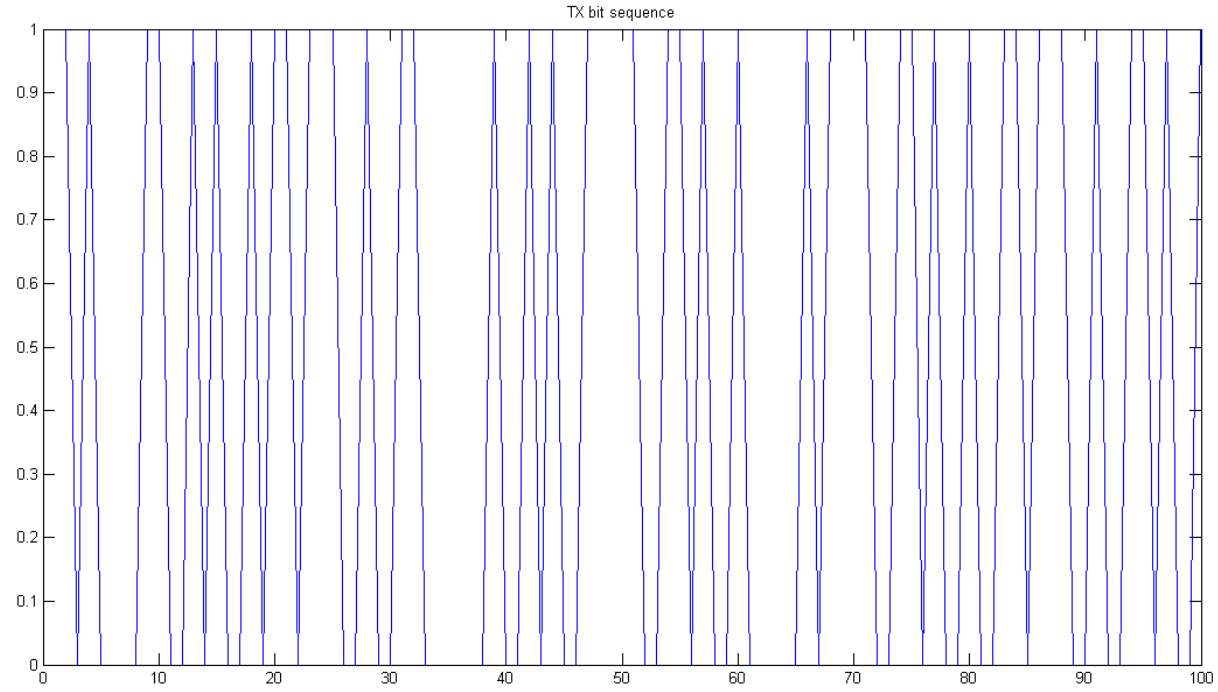

Fig. 5.3: Random bit stream representation.

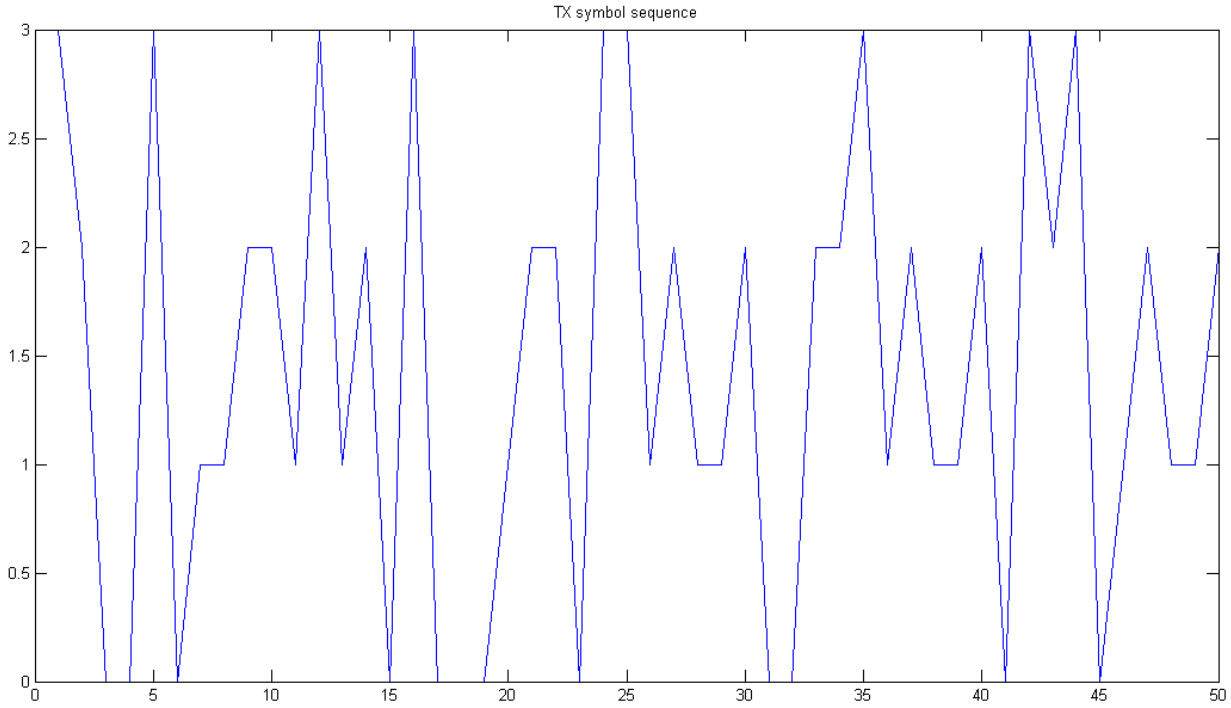

Fig. 5.4: Symbol stream representation of the figure 5.3 bit stream.

The transmitted signal is obtained transforming each symbol into his corresponding modulated impulse as can be seen in the figure 5.5. When the signal is received, it has been corrupted with noise. If the transmitted signal of the figure 5.5 is compared with the received signal with noise of the figure 5.6, it can be appreciated the "destruction" that the signal suffers.

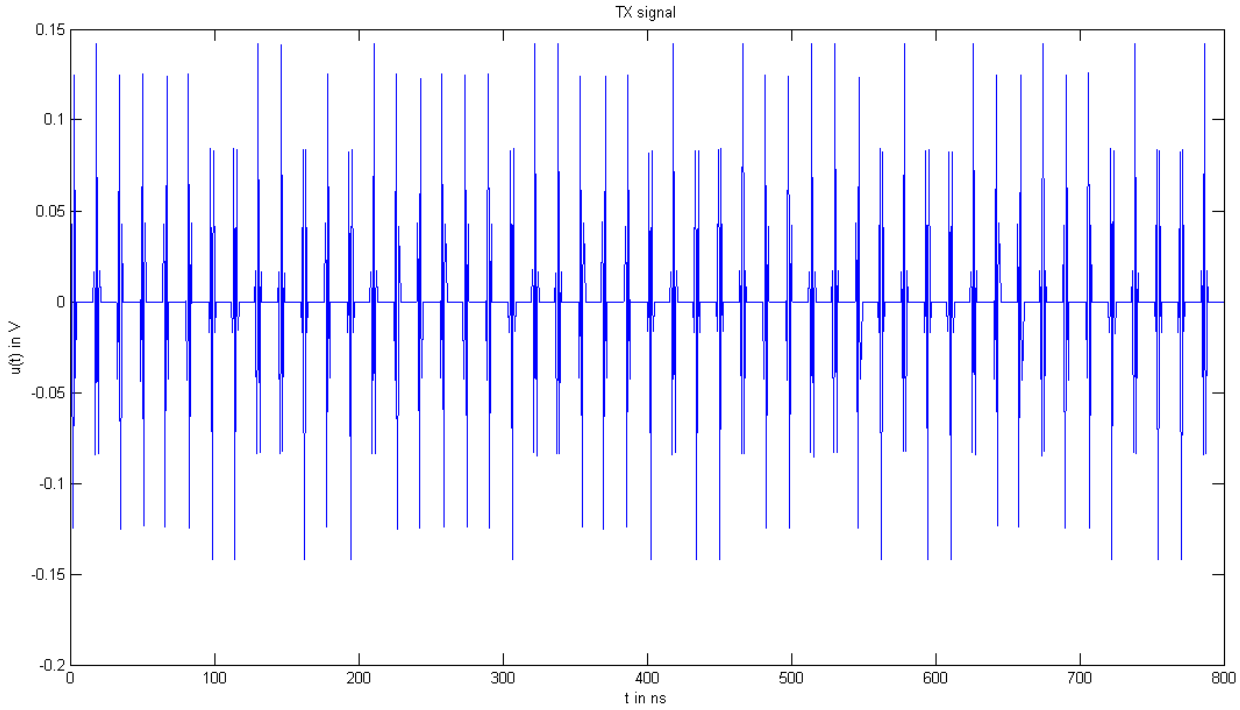

Fig. 5.5: Transmitted signal.

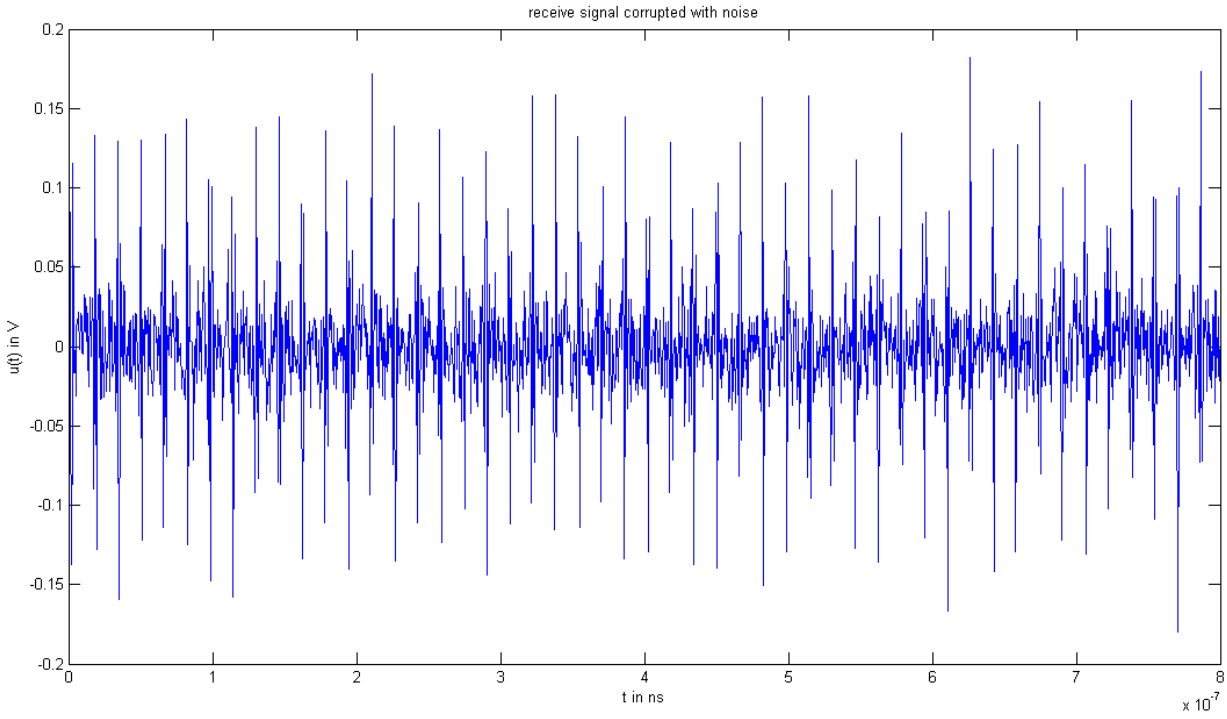

Fig. 5.6: Received signal with noise.

The received signal is filtered using different matched filter templates as it was explained in the previous sections. In this case, with four symbols available, there are four matched filter outputs, one by each symbol. They are depicted in the figures 5.7, 5.8, 5.9, 5.10.

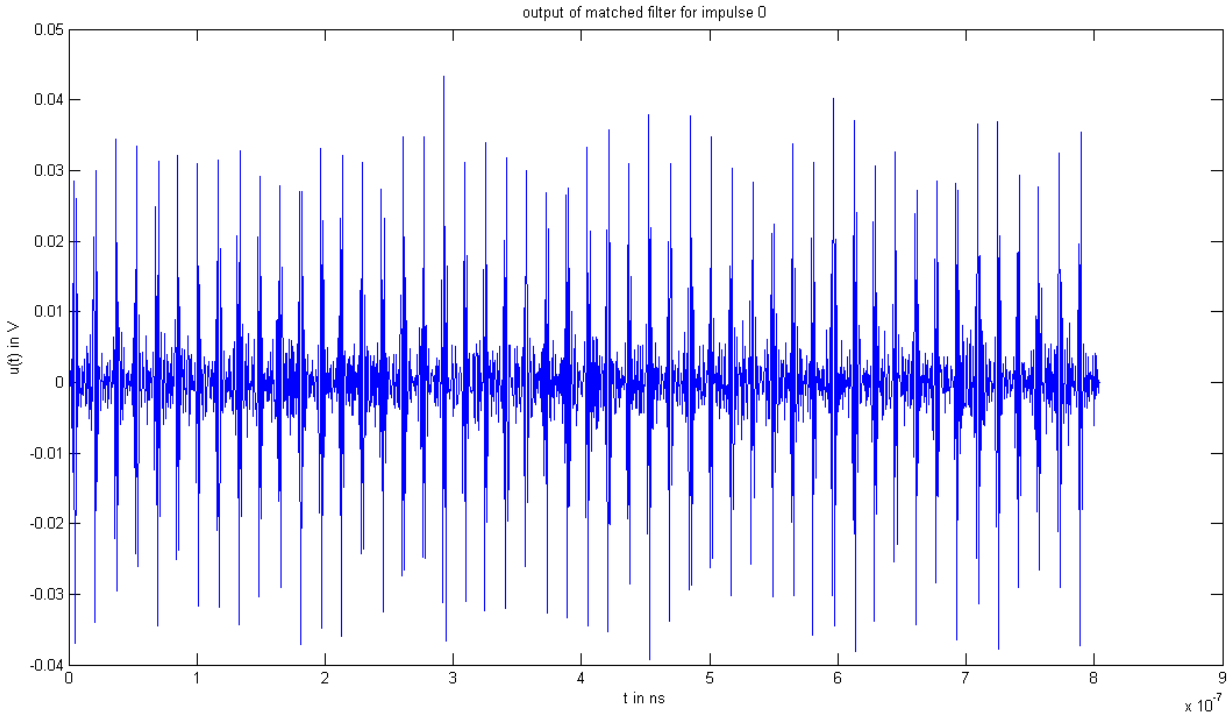

Fig. 5.7: Matched filter output for the symbol ‘0’ template.

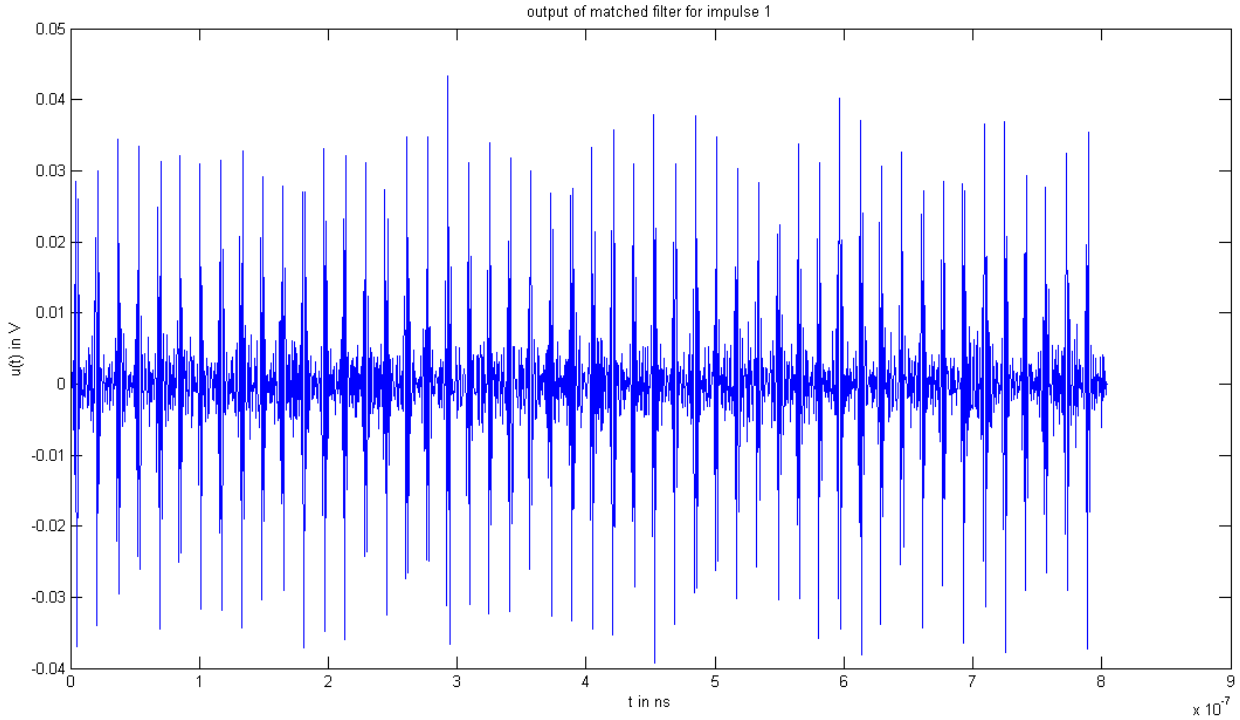

Fig. 5.8: Matched filter output for the symbol ‘1’ template.

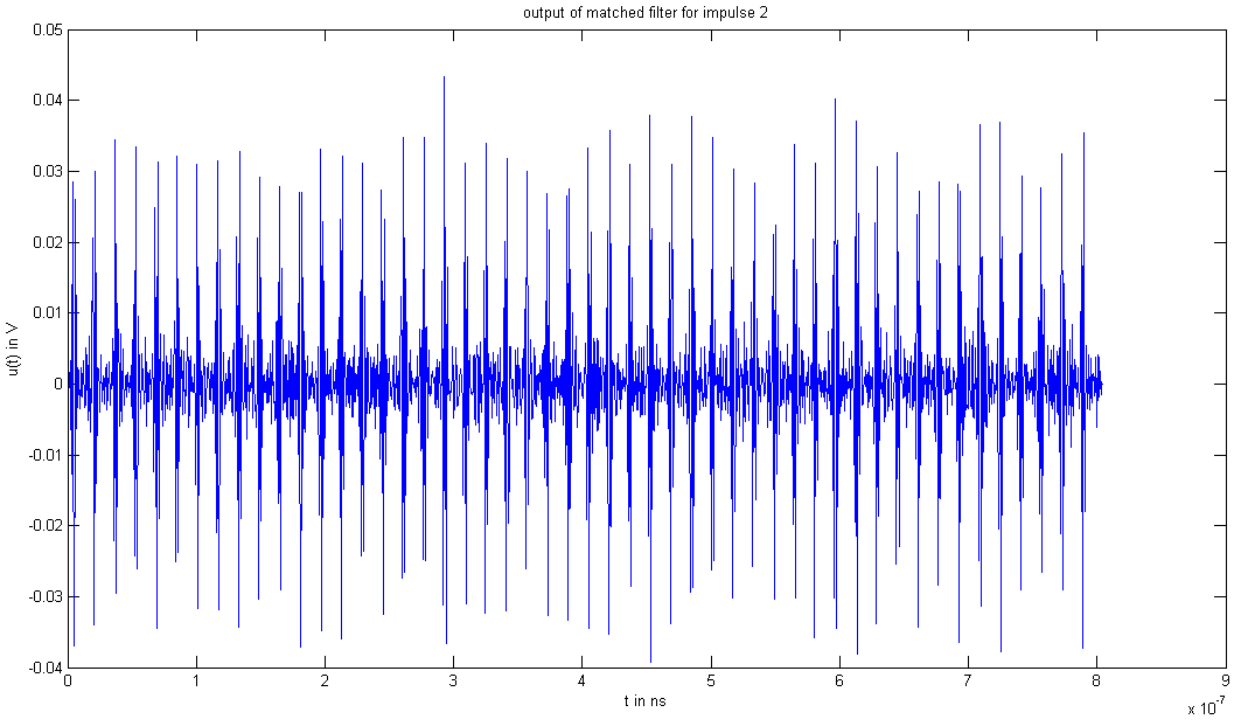

Fig. 5.9: Matched filter output for the symbol ‘2’ template.

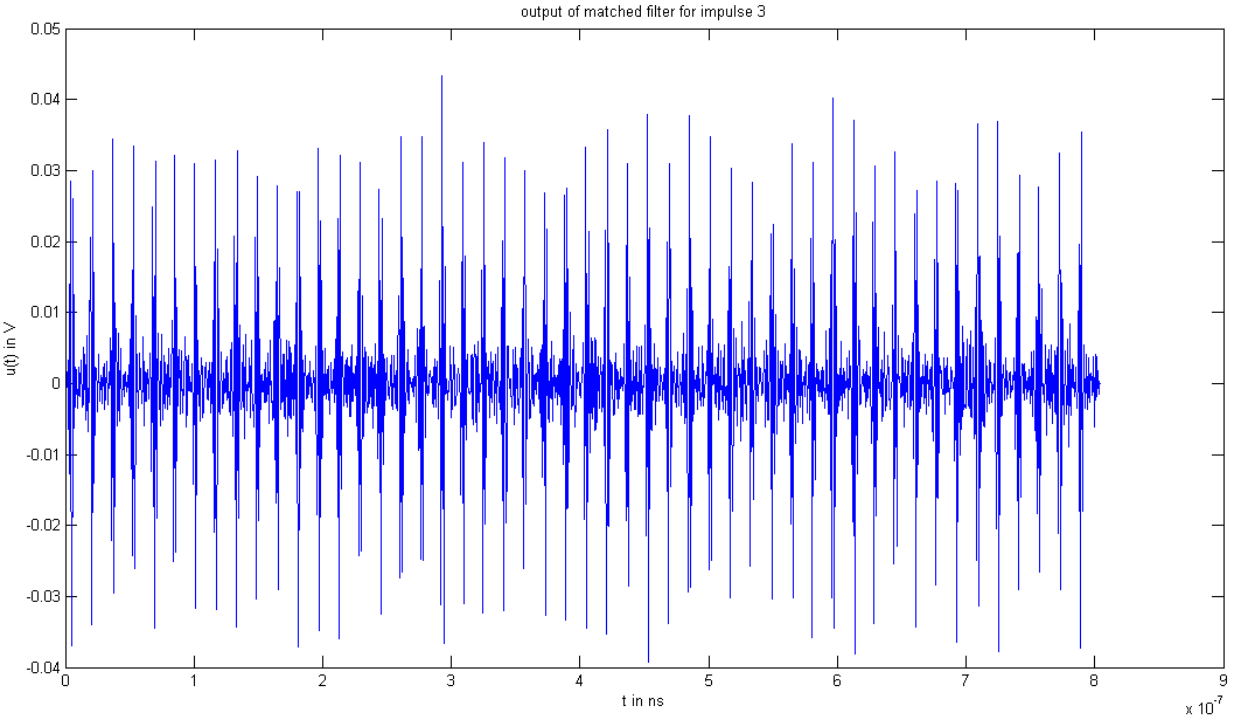

Fig. 5.10: Matched filter output for the symbol ‘3’ template.

As it was aforementioned, a sample of each matched filter output is taken for each symbol transmitted with f_{sym} frequency. Then, the samples taken from the same symbol are compared and the symbol decided is the one with highest matched filter output. In this process, the synchronization is very important and on it depends the successful of the receiver. The figures 5.11/5.12 and 5.13/5.14 show two situations where the synchronization is wrong. As can be seen, the transmitted signal and the received signal are completely different causing very high bit and symbol error rate.

On the contrary, if the synchronization time is good enough, the received signal can be exactly the same than the transmitted signal as is depicted in the figures 5.15/5.16.

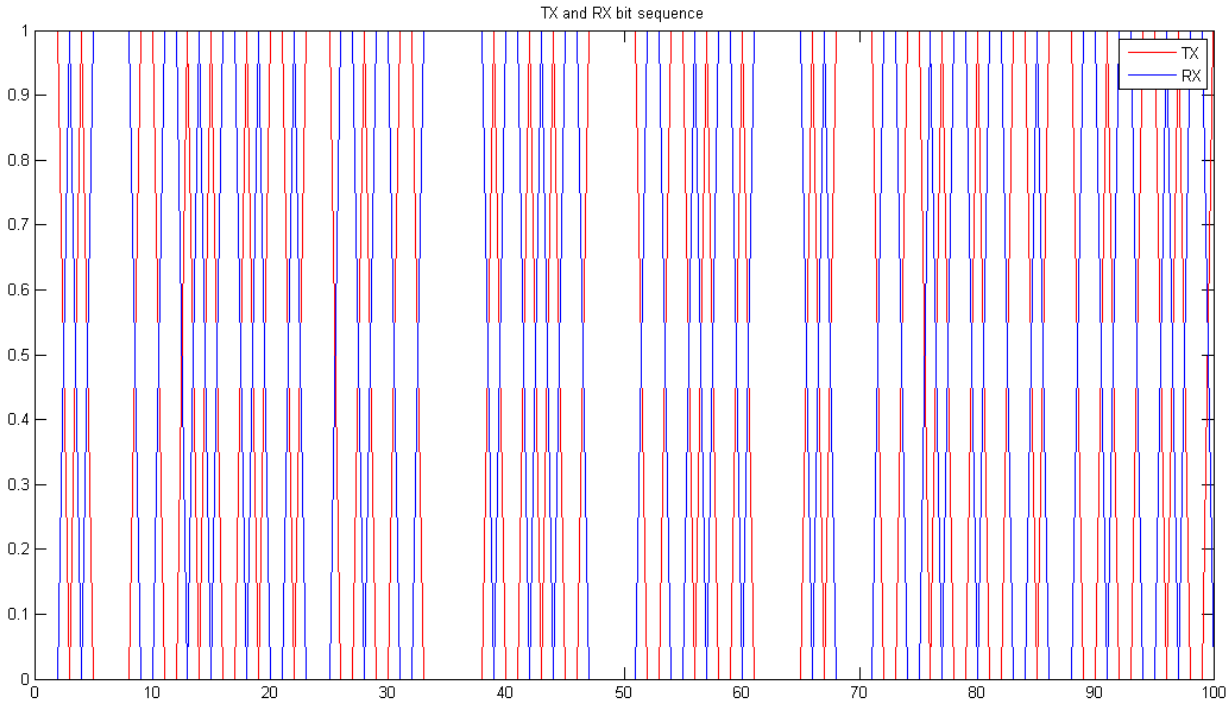

Fig. 5.11: Comparison between transmitted bit stream signal and received bit stream signal with a wrong synchronization 1.

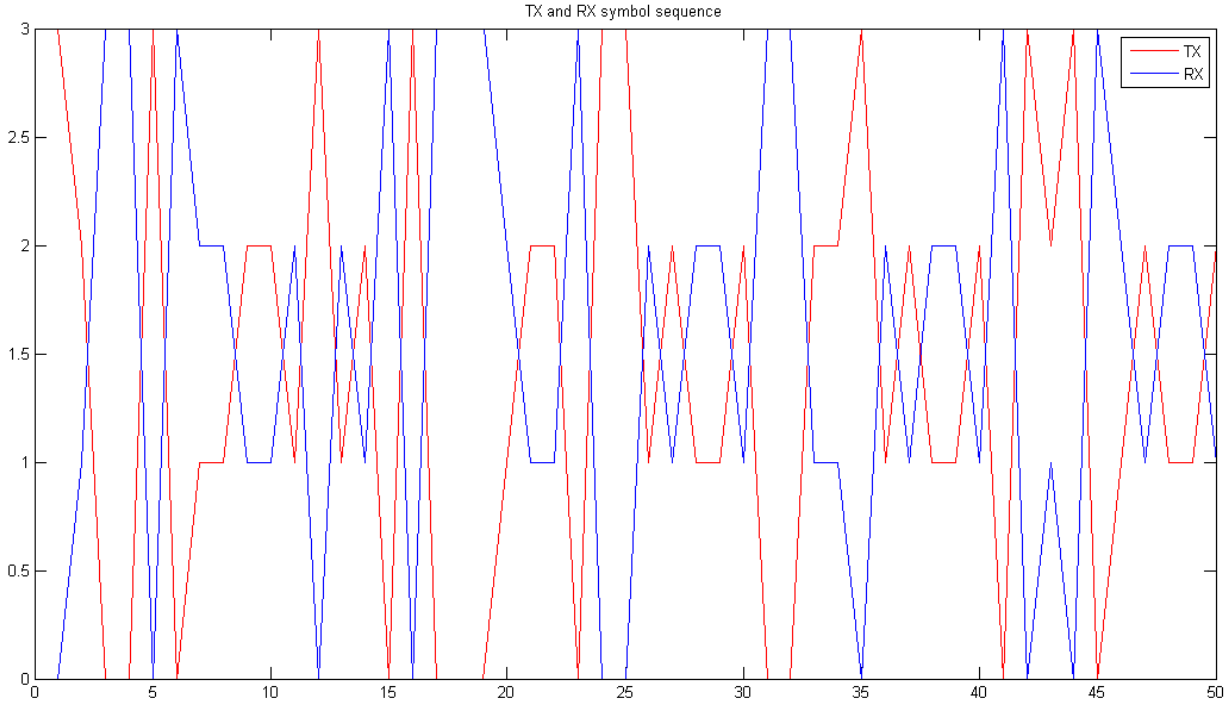

Fig. 5.12: Comparison between transmitted symbol stream signal and received symbol stream signal with a wrong synchronization 1.

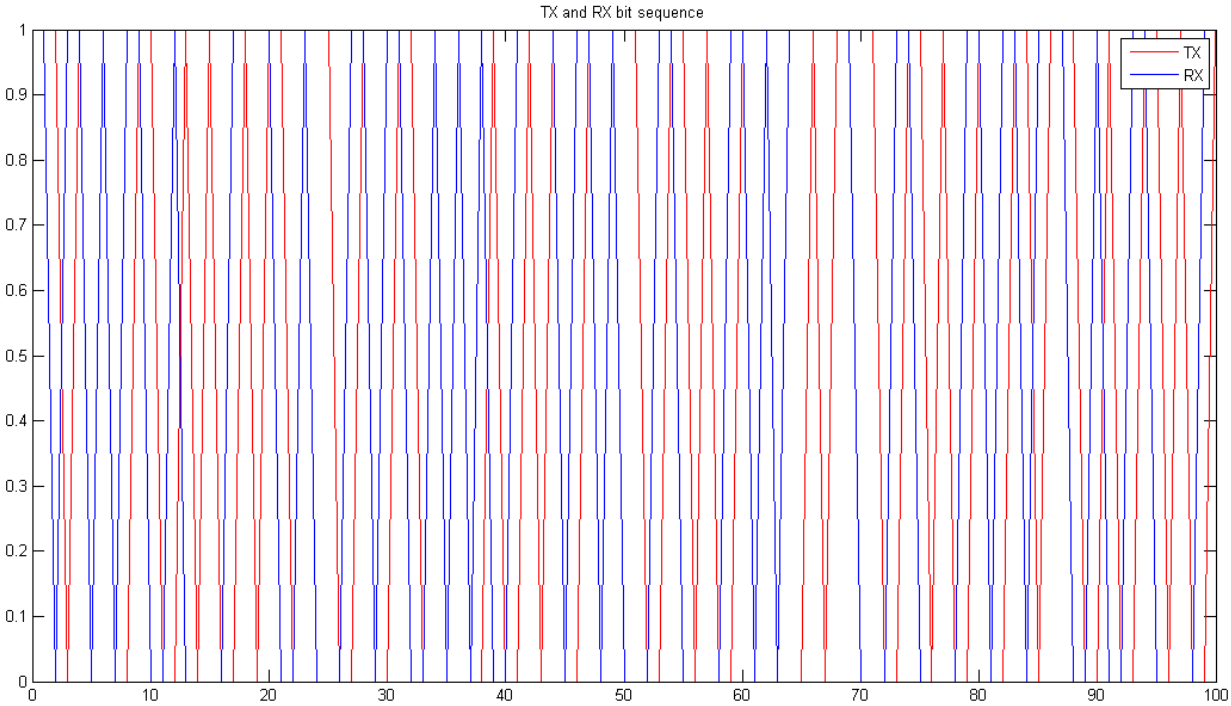

Fig. 5.13: Comparison between transmitted bit stream signal and received bit stream signal with a wrong synchronization 2.

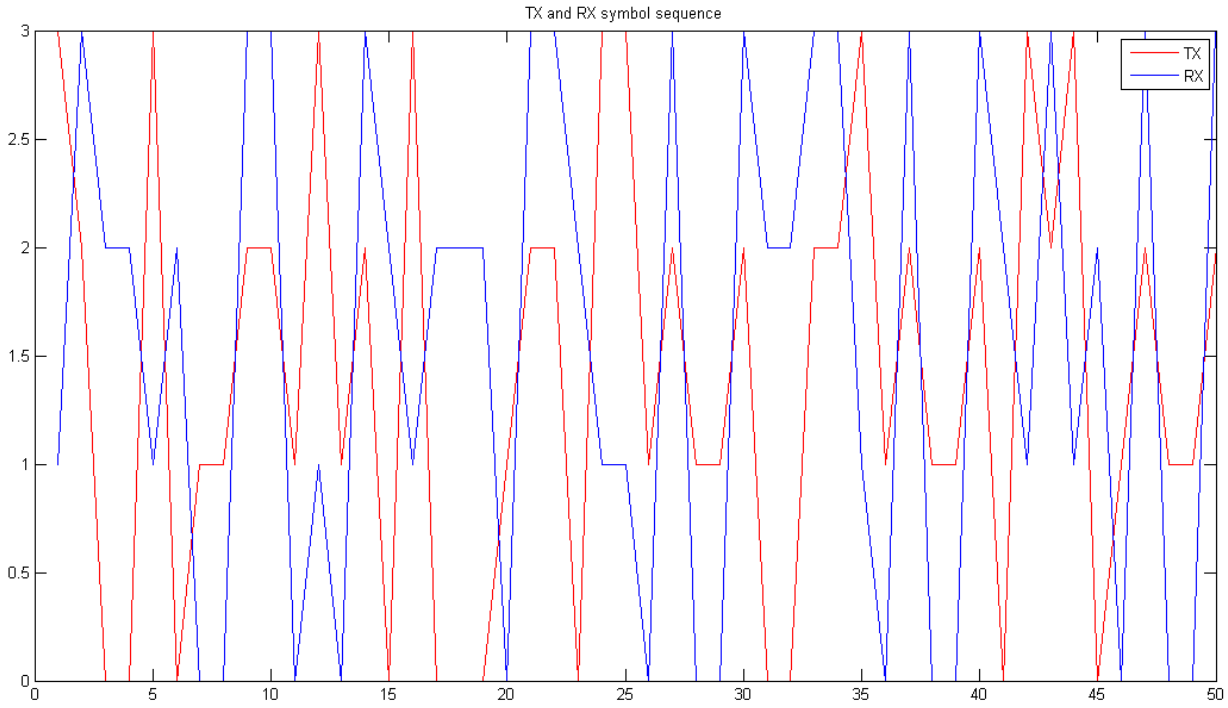

Fig. 5.14: Comparison between transmitted symbol stream signal and received symbol stream signal with a wrong synchronization 2.

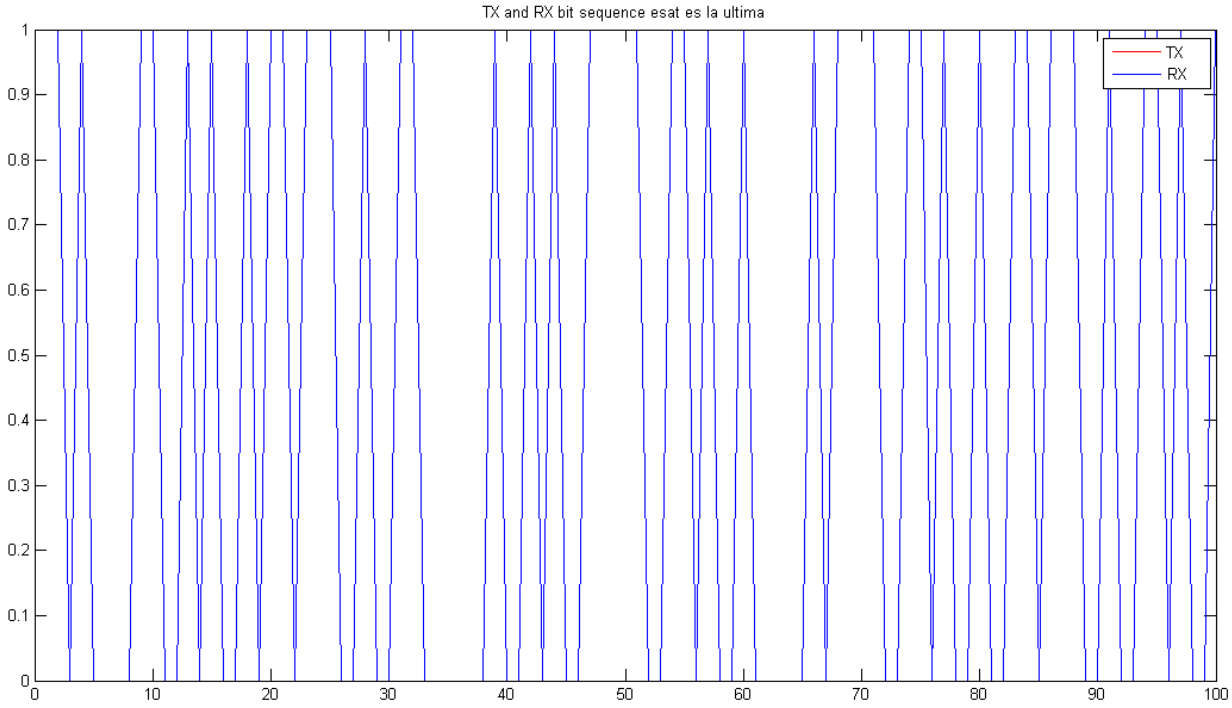

Fig. 5.15: Comparison between transmitted bit stream signal and received bit stream signal with a perfect synchronization.

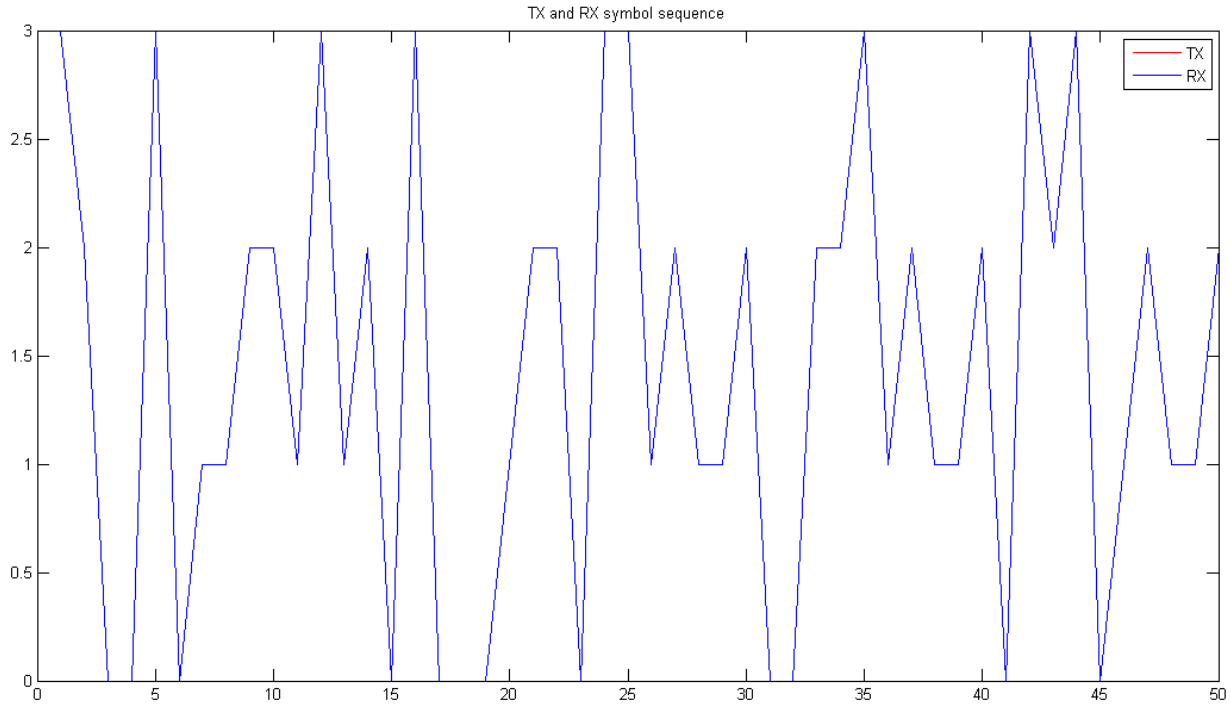

Fig. 5.16: Comparison between transmitted symbol stream signal and received symbol stream signal with a perfect synchronization.

As the main objective of this research is to improve the time needed to demodulate the signal, here there is a comparison of the computational time required between C and Matlab. The graphic shown in the figure 5.17 exhibits the different computational time employed by Matlab and C for different data length. At a glance can be seen that C code is much faster than Matlab code and as the data length is increasing the difference gets much higher. So, for a 100000 data length is gained a difference of 3 seconds and this difference shoots off until almost 30 seconds with a 1000000 data length. The graphic also exhibits an exponential shape as the data length is increasing.

Apart from the speed of the code, there is another aspect that must be considered. It is the bit and symbol error rates. They must keep approximately the same ranges that can be obtained in the Matlab routine. In order to satisfy this requirement, the figures 5.18 and 5.19 show a comparison of the bit error rates that can be obtained using C and Matlab routines. The different rates are obtained modifying the SNR level. It is possible to state that the results are practically the same, so the C routine is a valid solution to make faster the computational time of this kind of receiver.

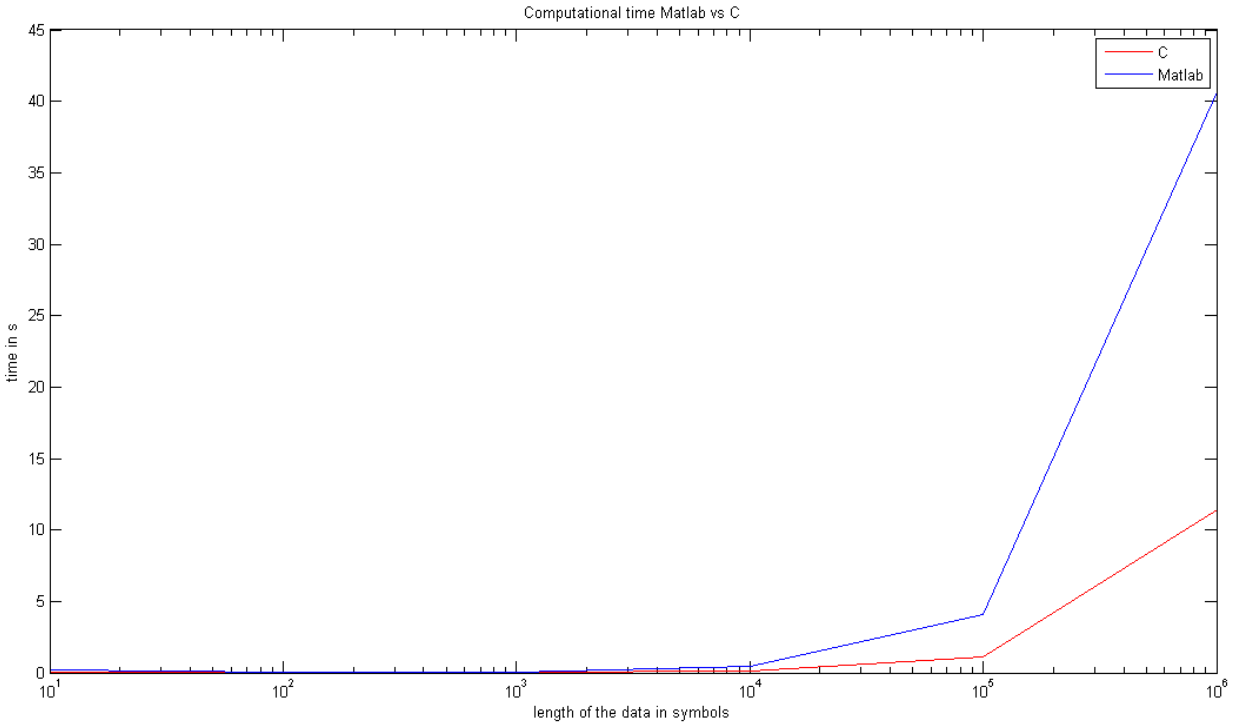

Fig. 5.17: Comparison of the computational time required between Matlab and C.

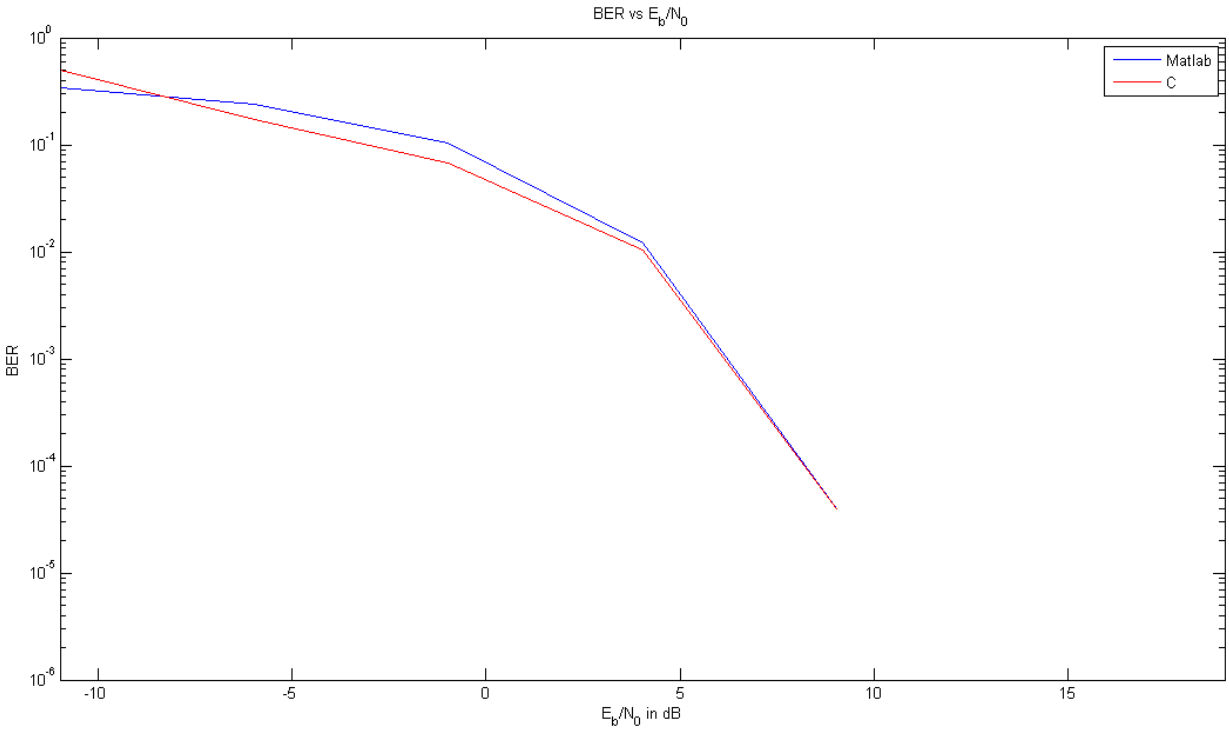

Fig. 5.18: Comparison of the BER obtained with C and Matlab for different E_b/N_0.

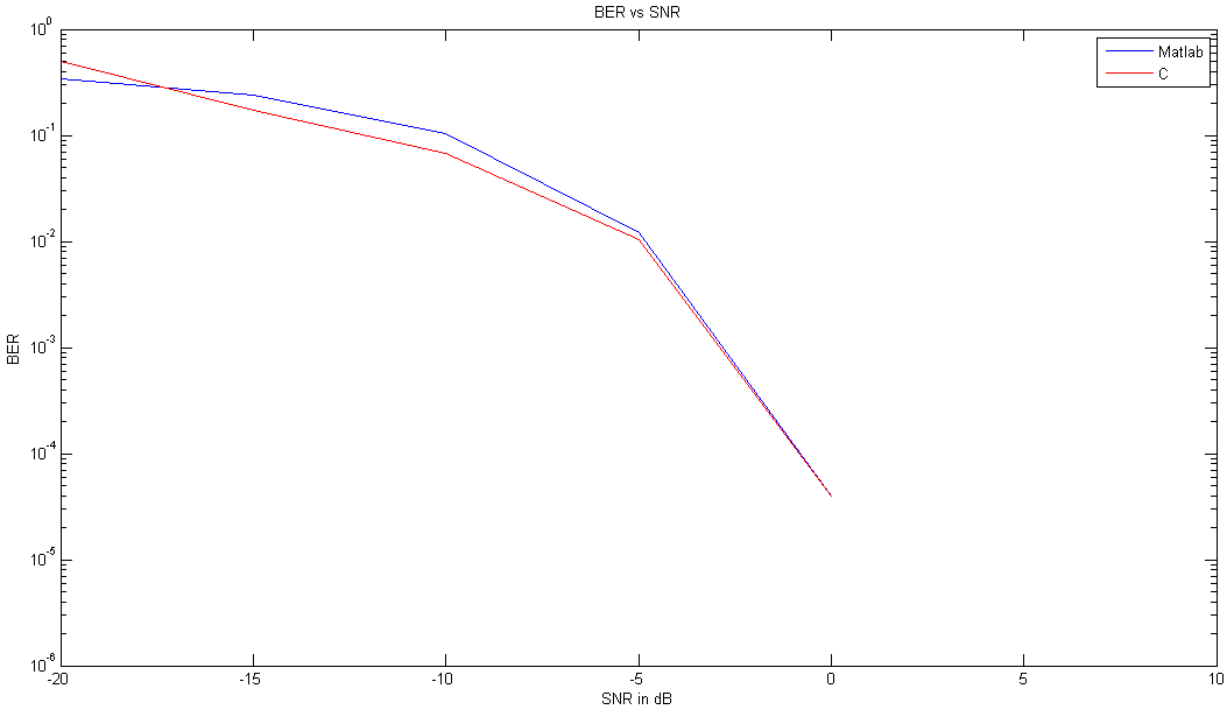

Fig. 5.19: Comparison of the BER obtained with C and Matlab for different *SNR*.

6. CONCLUSIONS

The main purpose of the present research was to implement a Software Defined UWB Impulse Receiver adapted to a Software Defined UWB transceiver capable to improve the computational times required. For that proposal, the software has been implemented in C language instead of Matlab, which was the language used for the transceiver implementation.

The fact of being Matlab a very high level programming language and therefore not suitable for real time applications made C language a very good candidate to reach the objective of this research. C is a low lever programming that can improve significantly the computational time. For integrating the receiver, programmed in C, with the tranceiver, implemented in Matlab, a mex-function has been used a link between both languages.

The explanation of this research has been structured in two distinct parts, a theorical design and a software design part. In the first part, the receiver has been explained from a theorical point of view. This explanation can be helpful for other students because different independent concepts have been explained in a simple way. The receiver exposition has been carried out describing the transceiver and transmitter architecture and the different concepts employed to reach the receiver architecture such a subsampling technique, bandpass sampling, matched filtering, PSK modulation and IR signals. All these concepts have been explained using equations, graphics and clarifying figures.

In the software design part, has been explained how to carry out the theorical explanation in a programming language. The receiver, mex-function and other complementary functions have been explained and described using flowcharts. Also, the time evolution signal from a simulation has been depicted in order to resume and understand the receiver process.

Different simulations have been carried out calculating the computational time required for Matlab and C routine. In these simulations, the data length has been changed second a logarithmic scale. As we aforementioned, the results obtained show that C routine can improve a 75% the time required to complete the reception process keeping the error rates.

Reducing the computational times can make possible to apply this kind of design to practical real time implementations. This concept, IR systems can also become a real alternative to OFDM systems due to quality of the radio link obtained which has a very high speed as required for the new communications services. The design using first order bandpass sampling avoid non-ideal mixer stages. This means that the possible practical implementations become simpler using less elements and also cheaper.

This software can also be helpful as a tool for other researches. The design can be modified in order to obtain different IR modulations. In this way, it would be possible to use this software to get measurements of the performance of several IR modulations in different scenarios.

7. BIBLIOGRAPHY AND REFERENCES

[1] G.R. Aiello and G.D. Rogerson, "Ultra-Wideband Wireless Systems," *IEEE Microwave Magazine*, vol. 4, pp. 36 – 47, June 2003.

[2] J. H. Reed et al., "*An Introduction to Ultra Wideband Communication Systems*," Prentice Hall, Upper Saddle River, NJ, USA, 2005.

[3] M. Ghaavami, L.B. Michael, and R. Kohno, "*Ultra Wideband Signals and Systems in Communication Engineering,*" 1st Edition, John Wiley & Sons, Ltd., Hoboken, NJ, USA, 2005.

[4] S. –W.M. Chen and R.W. Brodersen, "A subsampling Radio architecture for Ultrawideband Communications," *IEEE Transactions on Communications*, vol. 55, no. 10, pp. 5018-5031, Oct. 2007.

[5] Y.-L. Chao and R.A. Scholtz, "Ultra-Wideband Transmitted Reference Systems," *IEEE Transactions on Vehicular Technology*, vol. 54, no. 5, Sept. 2005.

[6] S. Gezici, F. Tufvesson, A.F. Molisch, "On the Performance of Transmitted-Reference Impulse Radio", *IEEE Global Telecommunications Conference GLOBECOM '04*, vol. 5, pp. 2874 – 2879, Nov./Dec. 2004.

[7] Blech, M. D.; Neumaier, P.; Ott, A. T.; Zan, A. A.; Eibert, T. F."Perfomance Analysis of a Software Defined Subsampling Ultra-Wideband B-/QPSK Impulse Radio Transceiver". Institut für Hochfrequenz-technik, Universität Stuttgart.

[8] S.-W. M. Chen and R. W. Brodersen, "A Subsampling Radio Architecture for Ultrawideband Communications," *IEEE Trans. Signal Proc.*, vol. 55, no.10, pp. 5018-5031, Oct. 2007.

[9] R.G. Vaughan et al., "The Theory of Bandpass Sampling," *IEEE Trans. Signal Proc.*, vol. 39, no. 9, pp. 1973-1984, Sept 1991.

[10] A. J. Coulson et al., "Frequency-Shifting using Bandpass Sampling," *IEEE Trans. Signal Proc.*, vol. 42, no.6, pp.1556-1559, June 1994.

[11] M. D. Blech et al.,"Concept of an UWB Impulse Radio B-QPSK Transmitter Based on Standard Logic Components", in *Proc. IEEE ICUWB 2008*, Hannover, Germany, Sep. 2008, pp. 225-228.

[12] J. G. Proakis, Digital Communications, MCGraw-Hill, New York, NY, 2001.

[13] The MathWorks. *"MEX-Files Guide"*. Available from: http://www.mathworks.com/support/tech-notes/1600/1605.html [Accesed from January to March].

[14] Laska, J., 2004. *"Writing C Functions in MATLAB (MEX-Files)"*. Conexions. Available from: http://cnx.org/content/m12348/latest/ [Accesed from January to March].

[15] The MathWorks. *"C and Fortran API Reference"*. Available from: http://www.mathworks.com/access/helpdesk/help/techdoc/index.html?/access/helpdesk/help/techdoc/apiref/bqoqnz0.html&http://www.mathworks.com/support/tech-notes/1600/1605.html#bqoqobe-1 [Accesed from January to March].

8. LIST OF FIGURES AND TABLES

9. C CODE FILE

```
#include "math.h"
#include "mex.h"
#include "matrix.h"
#include "stdio.h"

void mexFunction();
void qpsk_receiver();
void tukeywin();
void add_awgn();
double randn();
void conv();
double maxi();
void abs_complex();


void mexFunction(int  nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]) {
/*
Function to allow the communication between Matlab language and C language.
This function receive the calls from matlab and translate it to C.

INPUT PARAMETERS
 f_s                        sampling frequency
 n_samples_symbol           number of samples per symbol
 f_c                        carrier frequency
 b z_c                      bandwidth of gaussian impulse in the baseband
 p_max                      characteristic impedance
 g_t                        maximum allowed transmit power in 1 MHz
                            min. amplitude of impulse (see ICUWB2008 paper Blech ↙
     et al.)
 g_f                        attenuation at corner frequency (see ICUWB2008 paper ↙
     Blech et al.)
 tau                        time constant (see ICUWB2008 paper Blech et al.)
 sigma_b                    standard deviation of transmitted pulse train (see   ↙
    ICUWB2008 paper Blech et al.)
 mu_b                       mean value of transmitted pulse train (see ICUWB2008 ↙
     paper Blech et al.)
 K                          scale factor (see ICUWB2008 paper Blech et al.)
 lambda_mod                 phase modulation constant
 m_psk                      m=2: BPSK, m=4: QPSK, and so on up to m=16
 n_sym                      number of symbols to be generated
 rec_n_sym                  number of symbols for best timing
 sigma_jit                  rms_jitter
 tx_bit                     Bits transmitted
 tx_sym                     Symbols transmitted
 rx_in                      Noise

OUTPUT PARAMETERS
 rx_bit_tot                 Bits received
 rx_sym_tot                 Symbols received
*/

//Inputs
    double f_s, f_c, b, z_c, p_max, g_t, g_f, tau, sigma_b, mu_b, K, lambda_mod, ↙
     rec_sigma_noise,  snr, sigma_jit, channel_d, n_sym, rec_n_sym;
    int scenario, n_samples_symbol,  m_psk, long_bit;
    double *tx_bit, *tx_sym, *rx_in, *h_bp;

//Outputs
    double *rx_bit_tot;
    double *rx_sym_tot;


// Check for proper number of arguments
    if(nrhs != 26) {
        mexErrMsgTxt("Error  in number of inputs.");
    } else if (nlhs != 2) {
```

```
 62          mexErrMsgTxt("Error  in number  of outputs");
 63      }
 64
 65 // Assign pointers to each input
 66      f_s=*mxGetPr(prhs[0]);
 67      n_samples_symbol=*mxGetPr(prhs[1]);
 68      f_c=*mxGetPr(prhs[2]);
 69      b=*mxGetPr(prhs[3]);
 70      z_c=*mxGetPr(prhs[4]);
 71      p_max=*mxGetPr(prhs[5]);
 72      g_t=*mxGetPr(prhs[6]);
 73      g_f=*mxGetPr(prhs[7]);
 74      tau=*mxGetPr(prhs[8]);
 75      sigma_b=*mxGetPr(prhs[9]);
 76      mu_b=*mxGetPr(prhs[10]);
 77      K=*mxGetPr(prhs[11]);
 78      lambda_mod=*mxGetPr(prhs[12]);
 79      m_psk=*mxGetPr(prhs[13]);
 80      rec_sigma_noise=*mxGetPr(prhs[14]);
 81      snr=*mxGetPr(prhs[15]);
 82      sigma_jit=*mxGetPr(prhs[16]);
 83      channel_d=*mxGetPr(prhs[17]);
 84      n_sym=*mxGetPr(prhs[18]);
 85      rec_n_sym=*mxGetPr(prhs[19]);
 86      scenario=*mxGetPr(prhs[20]);
 87      tx_bit=mxGetPr(prhs[21]);
 88      tx_sym=mxGetPr(prhs[22]);
 89      rx_in=mxGetPr(prhs[23]);
 90      h_bp=mxGetPr(prhs[24]);
 91      long_bit=*mxGetPr(prhs[25]);
 92
 93 //Create vector for the output data
 94      plhs[0] = mxCreateDoubleMatrix(1,n_sym*long_bit, mxREAL);
 95      plhs[1] = mxCreateDoubleMatrix(1,n_sym,  mxREAL);
 96
 97 //Assign pointers to each output
 98      rx_bit_tot  = mxGetPr(plhs[0]);
 99      rx_sym_tot  = mxGetPr(plhs[1]);
100
101      qpsk_receiver(f_s,  n_samples_symbol,  f_c, b, z_c, p_max, g_t, g_f, tau,      ↙
         sigma_b, mu_b, K, lambda_mod, m_psk, rec_sigma_noise, snr, sigma_jit,          ↙
         channel_d, n_sym, rec_n_sym, scenario, tx_bit, tx_sym, rx_in, h_bp,            ↙
         rx_bit_tot, rx_sym_tot);
102
103 }
104
105 void abs_complex(double*  vector_real, double* vector_imag, int longitud, double* ↙
          result){
106      /* Function that calculates the absolute value of a vector of complex        ↙
         numbers.
107      vector_real = vector with the real part of the complex numbers
108      vector_imag = vector with the imaginary part of the complex numbers
109      longitud = lenght of the complex vector
110      */
111
112      int i;
113      for(i=0; i<longitud; i++){
114          result[i]=sqrt(pow(vector_real[i],2)+pow(vector_imag[i],2));
115      }
116 }
117
118 double randn(void){
119 /*
120  * Function that retuns an approximation of a normally distributed random        ↙
         numbers
121  * with unit variance
```

```
122  *
123  * division factor: randommax * sqrt(nrand / 12)
124  * with nrand = 16 and randommax = 0x1000
125  */
126         int sum;
127         int i;
128         sum=0;
129
130         for (i = 0; i < 16; i++)
131                 sum += rand() & 0xfff;
132         return (sum - 0x8000) * (1.0 / 4729.7);
133 }
134
135 void add_awgn(double* rx_in, double snr, int long_rx, double* final_signal){
136 /*Function for adding gaussian white noise
137  rx_in = signal input
138  long_rx = lenght of signal input
139  snr = signal to noise ratio
140  final_signal= signal result
141
142  Pnorm = sum(Vector.*Vector)/length(Vector)/system_impedance/sampling interval
143 */
144     double Pnorm;
145     double Prausch;
146     double *Rauschen;
147     int i;
148
149     Rauschen=(double  *)calloc (long_rx,sizeof(double));
150
151     Pnorm=0;
152     Prausch=0;
153
154     //calculation of the normalized power of the sequence
155     for(i=0; i<long_rx; i++){
156         Pnorm=Pnorm+(rx_in[i]*rx_in[i]);
157     }
158     Pnorm=Pnorm/long_rx/50/pow(5e-12,2);
159     //calculation of the normalized noise power of the sequence of an SNR
        (signal to noise ratio)
160     Prausch=Pnorm/(pow(10,(snr/10)));
161     //Generation of the noise sequence
162     for(i=0; i<long_rx; i++){
163         Rauschen[i]=0.4*randn()*sqrt(Prausch*50*pow(5e-12,2));
164         final_signal[i]=rx_in[i]+Rauschen[i];
165     }
166
167     free(Rauschen);
168 }
169 void tukeywin(int L, double r, double* window){
170 /*Function to create a tukey window
171  L = length of the window
172  r = ratio of taper (0 < r < 1)
173  window = result signal
174 */
175
176     int i;
177     double pi;
178
179     pi=3,1416;
180
181     if( r==0){
182         for(i=0; i<L; i++){
183         window [i]=1;
184         }
185     }
186     else{
```

```
187         for(i=0; i<L; i++){
188
189             if(0<=fabs(i-((L-1)/2))  && fabs(i-((L-1)/2))<=(r*(L-1)/2)){
190                 window[i]=1;
191             }else{
192                 window[i]=1/2*(1+cos(pi*(((i-(r*(L-1)/2))-(r*(L-1)/2))/(2*(1-r)* ↙
    ((L-1)/2)))));
193             }
194         }
195     }
196 }
197
198 void conv(double* v, int n, double* u, int m, double* result){
199 /* Function to create the convolution of two signals
200 v = signal to convolution
201 n = length of v
202 u = signal to convolution
203 m = length of m
204 result = result of the convolution
205 */
206
207 int j_init;
208 int j_fin;
209 int k;
210 int j;
211 double sum;
212
213     for(k=0; k<(m+n-1); k++){
214         sum=0;
215
216         if(1>=(k+1-n)+1)
217             j_init=0;
218         else
219             j_init=k+1-n;
220
221         if(k<=(m-1))
222             j_fin=k;
223         else
224             j_fin=m-1;
225
226         for(j=j_init; j<=j_fin; j++){
227             sum=sum+(u[j]*v[k-j]);
228         }
229         result[k]=sum;
230     }
231 }
232
233 void fliplr(double* vector, int n_v,double* invert){
234 /* Function to invert a signal
235 vector = signal to invert
236 n_v = length of vector-1
237 invert = inverted signal
238 */
239     int i;
240
241     for(i=0; i<n_v; i++){
242         invert[i]=vector[(n_v-1-i)];
243     }
244 }
245
246 double maxi(double *vector, int n_v){
247     /* Funtion to get the maximun value of a vector
248     vector = input vector
249     n_v = length of vector
250
251     */
```

```
252     int i;
253     double max;
254 
255     max=0;
256     for(i=0; i<n_v; i++){
257         if(max < vector[i])
258             max=vector[i];
259     }
260     return max;
261 }
262 
263 
264 void qpsk_receiver(double  f_s, int n_samples_symbol,  double f_c, double b,
        double z_c, double p_max, double g_t, double g_f, double tau, double sigma_b
        , double mu_b, double K, double lambda_mod,  int m_psk, double
        rec_sigma_noise,  double snr, double sigma_jit, double channel_d,  double
        n_sym, double rec_n_sym,  int scenario, double* tx_bit, double* tx_sym,
        double* rx_in, double* h_bp, double* rx_bit_total,  double* rx_sym_total){
265 
266 /*-------------------------------------------------------------------------
267   Variables
268   -------------------------------------------------------------------------
269 
270  f_s=2.000e9;                     %sampling frequency
271  n_samples_symbol=1*32;             %number of samples per symbol
272                   f_sym=f_s/n_samples_symbol;     %symbol frequency
273  f_c=500e6;                       %carrier frequency
274  b=500e6;                         %10 dB bandwidth of gaussian impulse in the
        baseband
275 
276  z_c=50;                          %characteristic impedance, usually 50 Ohms
277  p_max=1e-3*10^(-41.3/10);        %maximum allowed transmit power in 1 MHz
278  g_t=0.1;                         %min. amplitude of impulse (see ICUWB2008 paper
         Blech et al.)
279  g_f=10^(-10/20);                 %attenuation at corner frequency (see ICUWB2008
         paper Blech et al.)
280  tau=2*sqrt(log(g_t)*log(g_f))/(b*pi);%time constant (see ICUWB2008 paper Blech
        et al.)
281  sigma_b=1;                       %standard deviation of transmitted pulse train
        (see ICUWB2008 paper Blech et al.)
282  mu_b=0;                          %mean value of transmitted pulse train (see
        ICUWB2008 paper Blech et al.)
283  K=sqrt((-2*log(g_t)*p_max*z_c)/(pi*tau^2*(sigma_b^2*f_sym*1e6+mu_b^2*f_sym^2)))
        ;%scale factor (see ICUWB2008 paper Blech et al.)
284  lambda_mod=0;%pi/4;                %phase modulation constant
285  m_psk=4;                         %m=2: BPSK, m=4: QPSK, and so on up to m=16
286 
287  n_sym=1000;                        %number of symbols to be generated
288 
289  if(n_sym>=100)
290      rec_n_sym=100;               %number of symbols for best timing
291  else
292      rec_n_sym=n_sym;             %number of symbols for best timing
293  end;
294  snr_start=20;                   %start of snr for sweep
295  snr_step=5;                      %step of snr for sweep
296  snr_stop=20;                     %stop of snr for sweep
297 
298  sigma_jit=0;                     %rms_jitter
299  sigma_jit_start=1e-12;           %start of rms_jitter for sweep
300  sigma_jit_step_factor=10;        %step of rms_jitter for sweep
301  sigma_jit_stop=1e-12;          %stop of rms_jitter for sweep
302 
303  rec_full_scale=0.04;             %full scale voltage of the AD converter (8
        times the vertical scale/div)
304  rec_n_res=8;                     %nominal resolution in bit of the AD converter
```

```
      used in the receiver
305
306
307  flag_measurement=0;            %switches between measured and simulated input ✔
      data
308  flag_channel=0;                %switches channel on and off
309  flag_equalizer=0;              %switches equalizer on and off
310  flag_figures=0;                %switches figures on and off
311  flag_results=0;                %switches result figures on and off
312
313
314  tx_bit                         Bits transmitted
315  tx_sym                         Symbols transmitted
316  rx_in                          Noise
317
318  rx_bit_totat                   Bits received
319  rx_sym_totat                   Symbols received
320 */
321
322 //-------------------------------------------------------------------------
323 // Variables
324 //-------------------------------------------------------------------------
325
326
327 //For LUTs for Gray-Code
328     const float pi=3.14159;
329     double** gray_angle;
330
331 //For BaseBand Impulse
332     double *pulse_bb;
333     double t;
334     double pulse_help;
335     double *pulse_help1;
336     double *pulse_help2;
337     double *window;
338     int i;
339     int n;
340     int m;
341     int nbb;
342
343 //For Modulated Impulses
344     double **pulse_mod;
345
346 //For Addition Noise
347     double *rx_in_noisy;
348
349 //For Matched Filter Receiver
350     int nmf;
351     double **pulse_mf;
352     double **mf_out;
353     double *pulse_mf_inv;
354
355 //For ML Decision (first rec_n_sym)
356     double **rx_bit;
357     double **rx_sym;
358     int index_offset;
359     int n_offset;
360     int k;
361     double *values;
362     double maximun;
363     double *help;
364
365 //For Computation SER and BER (first rec_n_sym)
366     double *n_SER;
367     double *SER;
368     double *n_BER;
```

```
369     double *BER;
370     double *Eb_N0;
371     double **results;
372     double BER_min;
373     int i_optimun;
374
375 //For ML Decision
376     double *rx_bit_tot;
377     double *rx_sym_tot;
378
379 //For Computation SER and BE
380     double n_SER_tot;
381     double SER_tot;
382     double n_BER_tot;
383     double BER_tot;
384     double Eb_N0_tot;
385
386
387 // ------------------------------------------------------------------------
388 // LUTs for Gray-Code
389 // ------------------------------------------------------------------------
390
391
392 //Gray angles
393     gray_angle  = (double  **)calloc  (16 , sizeof(double  *));
394
395     for (i=0; i<16; i++){
396         gray_angle[i]  = (double  *)calloc  (16,sizeof(double));
397     }
398
399     //BPSK
400     gray_angle[1][0]=0*pi/(m_psk/2);
401     gray_angle[1][1]=1*pi/(m_psk/2);
402
403     //QPSK
404     gray_angle[3][0]=0*pi/(m_psk/2);
405     gray_angle[3][1]=1*pi/(m_psk/2);
406     gray_angle[3][2]=3*pi/(m_psk/2)-2*pi;
407     gray_angle[3][3]=2*pi/(m_psk/2);
408
409     //8-PSK
410     gray_angle[7][0]=0*pi/(m_psk/2);
411     gray_angle[7][1]=1*pi/(m_psk/2);
412     gray_angle[7][2]=3*pi/(m_psk/2);
413     gray_angle[7][3]=2*pi/(m_psk/2);
414     gray_angle[7][4]=7*pi/(m_psk/2)-2*pi;
415     gray_angle[7][5]=6*pi/(m_psk/2)-2*pi;
416     gray_angle[7][6]=4*pi/(m_psk/2);
417     gray_angle[7][7]=5*pi/(m_psk/2)-2*pi;
418
419     //16-PSK
420     gray_angle[15][0]=0*pi/(m_psk/2);
421     gray_angle[15][1]=1*pi/(m_psk/2);
422     gray_angle[15][2]=3*pi/(m_psk/2);
423     gray_angle[15][3]=2*pi/(m_psk/2);
424     gray_angle[15][4]=7*pi/(m_psk/2);
425     gray_angle[15][5]=6*pi/(m_psk/2);
426     gray_angle[15][6]=4*pi/(m_psk/2);
427     gray_angle[15][7]=5*pi/(m_psk/2);
428     gray_angle[15][8]=15*pi/(m_psk/2)-2*pi;
429     gray_angle[15][9]=14*pi/(m_psk/2)-2*pi;
430     gray_angle[15][10]=12*pi/(m_psk/2)-2*pi;
431     gray_angle[15][11]=13*pi/(m_psk/2)-2*pi;
432     gray_angle[15][12]=8*pi/(m_psk/2)-2*pi;
433     gray_angle[15][13]=9*pi/(m_psk/2)-2*pi;
434     gray_angle[15][14]=11*pi/(m_psk/2);
```

```
435     gray_angle[15][15]=10*pi/(m_psk/2-2*pi);
436
437
438 //-------------------------------------------------------------------------
439 //Generation of baseband impulse
440 //-------------------------------------------------------------------------
441
442     //reserve space for the vectors
443     pulse_bb = (double *)calloc (2*n_samples_symbol,sizeof(double));
444     pulse_help1 = (double *)calloc (2*n_samples_symbol,sizeof(double));
445     pulse_help2 = (double *)calloc (2*n_samples_symbol,sizeof(double));
446
447
448     pulse_help=0;
449     t=0;
450
451     for(nbb=0; nbb<n_samples_symbol; nbb++){
452         pulse_help=K*exp(log(g_t)*pow((t/tau),2));
453         t=t+(1/f_s);
454         if(pulse_help>K*g_t)
455             pulse_help1[nbb]=pulse_help;
456         else{
457             pulse_help1[nbb]=pulse_help;
458             break;
459         }
460     }
461
462     fliplr(pulse_help1,nbb+1,pulse_help2);
463
464     for(m=0; m<nbb; m++){
465         pulse_bb[m]=pulse_help2[m];
466     }
467
468     for(m=nbb; m<=(2*nbb); m++){
469         pulse_bb[m]=pulse_help1[m-nbb];
470     }
471
472     window=(double *)malloc (((2*nbb)+1)*sizeof(double));
473     tukeywin(((2*n)+1),0.2,window);    //Generation of tukey window
474
475     for(m=0; m<=(2*nbb); m++){
476         pulse_bb[m]=pulse_bb[m]*window[m];
477     }
478
479     //Free memory
480     free(pulse_help1);
481     free(pulse_help2);
482     free(window);
483
484
485 //-------------------------------------------------------------------------
486 //Generation of modulated impulses
487 //-------------------------------------------------------------------------
488 //even bits represent MSB
489 //odd bits represent LSB
490 //--> Gray-Mapping
491 //constellation diagram with lambda
492
493
494     //reserve space for the vectors
495     pulse_mod = (double **)calloc (m_psk ,sizeof(double *));
496     i=0;
497     for (i=0; i<m_psk; i++){
498         pulse_mod[i] = (double *)calloc (2*n_samples_symbol,sizeof(double));
499     }
500
```

```
501     //generation of impulse
502     t=0;
503     for(n=0; n<(n_samples_symbol);  n++){
504         for(m=0; m<m_psk; m++){
505             pulse_mod[m][n]=pulse_bb[n]*cos((2*pi*f_c*t)+gray_angle[m_psk-1][m]+ ↙
        lambda_mod);
506         }
507         t=t+(1/f_s);
508     }
509
510 //-------------------------------------------------------------------------
511 //Addition of noise to input
512 //(channel model already contains noise, additional
513 //noise of the receiver can be added)
514 //-------------------------------------------------------------------------
515
516 //Reserve space
517 rx_in_noisy=(double  *)calloc (n_sym*n_samples_symbol,sizeof(double));
518
519 switch (scenario){
520     case 1:
521         add_awgn(rx_in,  snr, ((int)n_sym*n_samples_symbol),rx_in_noisy);
522         break;
523
524     case 2:
525         add_awgn(rx_in,  snr, ((int)n_sym*n_samples_symbol),rx_in_noisy);
526         break;
527
528     case 3:
529         rx_in_noisy = rx_in;            //with ideal LOS channel
530         break;
531
532     case 4:
533         rx_in_noisy = rx_in;            //with ideal LOS channel and antennas
534         break;
535
536     case 5:
537         rx_in_noisy = rx_in;            //with non-ideal LOS channel and        ↙
        antennas
538         break;
539
540     case 6:
541         rx_in_noisy = rx_in;            //like 3 with equalizer
542         break;
543
544     case 7:
545         rx_in_noisy = rx_in;            //like 4 with equalizer
546         break;
547
548     case 8:
549         rx_in_noisy = rx_in;            //like 5 with equalizer
550         break;
551
552     case 9:
553         rx_in_noisy = rx_in;            //NLOS with equalizer
554         break;
555
556     case 10:
557         rx_in_noisy = rx_in;            //measurement with equalizer
558         break;
559
560 }
561
562
563 //-------------------------------------------------------------------------
564 //Matched filter receiver
```

```
565 //---------------------------------------------------------------------------
566
567
568 //Getting length of pulse_mf
569     for(nmf=0; nmf<n_samples_symbol;  nmf++){
570         if((pulse_mod[0][nmf]==0)  && (pulse_mod[0][nmf+1]==0))
571             break;
572     }
573
574 //reserve space for the vectors
575     pulse_mf = (double **)calloc (m_psk , sizeof(double *));
576     i=0;
577     for (i=0; i<m_psk; i++){
578         pulse_mf[i] = (double *)calloc ((nmf+1),sizeof(double));
579     }
580
581
582 //Generation of Filter templates
583     for(n=0; n<=nmf; n++){
584         for(m=0; m<m_psk; m++){
585             pulse_mf[m][n]=pulse_mod[m][n];
586         }
587     }
588
589 //reserve space for the vectors
590     mf_out=(double **)calloc (m_psk, sizeof(double *));
591     for (i=0; i<m_psk; i++){
592         mf_out[i] = (double *)calloc (((n_sym*n_samples_symbol)+(nmf+1)-1),
    sizeof(double));
593     }
594
595 //correlation
596     pulse_mf_inv= (double *)calloc ((nmf+1),sizeof(double));
597     for (i=0; i<m_psk; i++){ fliplr(pulse_mf[i],(nmf+1),pulse_mf_inv);
598         conv(rx_in_noisy,(int)(n_sym*n_samples_symbol),pulse_mf_inv,(int)(nmf+1)
599
    ,mf_out[i]);
600     }
601
602
603
604 //---------------------------------------------------------------------
605 //ML decision
606 //and detection of best synchronization with first rec_n_sym
607 //---------------------------------------------------------------------
608
609 //reserve space for the vectors and initialize
610     rx_bit = (double **)calloc ((n_samples_symbol/4)  , sizeof(double *));
611     rx_sym = (double **)calloc ((n_samples_symbol/4)  , sizeof(double *));
612
613     for (i=0; i<(n_samples_symbol/4);  i++){
614         rx_bit[i] = (double *)calloc (rec_n_sym*(log(m_psk)/log(2)),sizeof
    (double));
615         rx_sym[i] = (double *)calloc (rec_n_sym,sizeof(double));
616     }
617
618     values = (double *)calloc (m_psk,sizeof(double));
619     help = (double *)calloc ((log(m_psk)/log(2)),sizeof(double));
620     n_SER = (double *)calloc (5,sizeof(double));
621     SER = (double *)calloc (5,sizeof(double));
622     n_BER = (double *)calloc (5,sizeof(double));
623     BER = (double *)calloc (5,sizeof(double));
624     Eb_N0 = (double *)calloc (5,sizeof(double));
625     results = (double **)calloc (5 , sizeof(double *));
626     for(i=0; i<5; i++){
627         results[i] = (double *)calloc (4,sizeof(double));
```

```
628     }
629
630     index_offset=0;
631
632     for(n_offset=-24;  n_offset<=-20;  n_offset++){
633         k=0;
634
635         for(n=32;n<=(33+(rec_n_sym-1)*n_samples_symbol);n=n+n_samples_symbol){
636             for(m=0;  m<m_psk;  m++){
637                 values[m]=mf_out[m][n+n_offset];
638             }
639             maximun=maxi(values,m_psk);
640
641             for(m=0;  m<m_psk;  m++){
642                 if(maximun==mf_out[m][n+n_offset]){
643                     rx_sym[index_offset][k]=m;
644                     k++;
645                     break;
646                 }
647             }
648         }
649
650         for(i=0;  i<(log(m_psk)/log(2));  i++)
651             help[i]=pow(2,i);
652
653         k=0;
654         for(n=0;  n<rec_n_sym;  n++){
655             for(i=0;  i<(log(m_psk)/log(2));  i++){
656                 rx_bit[index_offset][k]=((int)rx_sym[index_offset][n]  & (int)    ↙
    help[i])/help[i];
657                 k++;
658             }
659         }
660
661     //-------------------------------------------------------------------------
662     //Compute  SER and BER
663     //-------------------------------------------------------------------------
664
665         //symbol  error  ratio
666         n_SER[index_offset]=0;
667         for(n=0;  n<rec_n_sym;  n++){
668             if(tx_sym[n]!=rx_sym[index_offset][n])
669               n_SER[index_offset]=n_SER[index_offset]+1;
670         }
671         SER[index_offset]=n_SER[index_offset]/rec_n_sym;    //SER   output
672
673         //bit  error  ratio
674         n_BER[index_offset]=0;
675
676         for(n=0;  n<(rec_n_sym*(log(m_psk)/log(2)));  n++){
677             if(tx_bit[n]!=rx_bit[index_offset][n]){
678                 n_BER[index_offset]=n_BER[index_offset]+1;
679             }
680         }
681         BER[index_offset]=n_BER[index_offset]/(rec_n_sym*(log(m_psk)/log(2))); / ↙
    /BER   output
682
683         Eb_N0[index_offset]=snr+10*log10(b/(1/(n_samples_symbol*1/(f_s))));   //  ↙
    Eb/N0
684
685         //prepare  matrix  to  save  the  computed  results  in  a  file
686         results[index_offset][0]=snr;           //signal  to  noise  ratio
687         results[index_offset][1]=Eb_N0[index_offset];  //Eb/N0
688         results[index_offset][2]=SER[index_offset];    //symbol  error  ratio
689         results[index_offset][3]=BER[index_offset];    //bit  error  ratio
690
```

```
691         index_offset++;
692     }
693
694     //find best timing-offset
695     BER_min=rec_n_sym*(log(m_psk)/log(2));
696     for(i=0; i<5; i++){
697         if(BER[i]<BER_min){
698             BER_min=BER[i];
699             i_optimun=i;
700         }
701     }
702
703 //---------------------------------------------------------------------
704 //ML decision
705 //and detection of best synchronization
706 //repeat BER-Calculation with whole sequence and correct timing
707 //---------------------------------------------------------------------
708
709     //Free memory and initialization of receive bits and symbols
710     for (i=0; i<(n_samples_symbol/4);  i++){
711         free (rx_sym[i]);
712         free (rx_bit[i]);
713     }
714     free(rx_sym);
715     free(rx_bit);
716
717     rx_bit_tot  = (double *)calloc (n_sym*(log(m_psk)/log(2)),sizeof(double));
718     rx_sym_tot  = (double *)calloc (n_sym,sizeof(double));
719     index_offset=i_optimun;
720     n_offset =-24+i_optimun;
721
722     k=0;
723         for(n=32;n<=(33+(n_sym-1)*n_samples_symbol);n=n+n_samples_symbol){
724             for(m=0; m<m_psk; m++){
725                 values[m]=mf_out[m][n+n_offset];
726             }
727
728             maximun=maxi(values,m_psk);
729             for(m=0; m<m_psk; m++){
730                 if(maximun==mf_out[m][n+n_offset]){
731                     rx_sym_tot[k]=m;
732                     k++;
733                     break;
734                 }
735             }
736         }
737
738     //demapping of received symbols to bitstream
739     //precalculation of time intense computation
740     for(i=0; i<(log(m_psk)/log(2)); i++){
741         help[i]=pow(2,i);
742     }
743     k=0;
744     for(n=0; n<n_sym; n++){
745         for(i=0; i<(log(m_psk)/log(2)); i++){
746             rx_bit_tot[k]=((int)rx_sym_tot[n] & (int)help[i])/help[i];
747             k++;
748         }
749     }
750
751
752 //---------------------------------------------------------------------------
753 //Compute SER and BER
754 //---------------------------------------------------------------------------
755
756     //symbol error ratio
```

```
757     n_SER_tot=0;
758     for(n=0; n<n_sym; n++){
759         if(tx_sym[n] != rx_sym_tot[n]){
760             n_SER_tot++;
761         }
762     }
763     SER_tot=n_SER_tot/n_sym;   //SER output
764
765     //bit error ratio
766     n_BER_tot=0;
767
768     for(n=1; n<(n_sym*(log(m_psk)/log(2))); n++){
769         if(tx_bit[n]!=rx_bit_tot[n]){
770             n_BER_tot++;
771         }
772     }
773     BER_tot=n_BER_tot/(n_sym*(log(m_psk)/log(2))); //BER  output
774     Eb_N0_tot=snr+10*log10(b/(1/(n_samples_symbol*1/(f_s))));   //Eb/N0
775
776 //-------------------------------------------------------------------------
777 //Return signals and free memory
778 //-------------------------------------------------------------------------
779
780     //Copy rx_bit and rx_sym
781     for (i=0; i<n_sym*(log(m_psk)/log(2)); i++){
782         rx_bit_total[i]=rx_bit_tot[i];
783     }
784     for (i=0; i<n_sym; i++){
785         rx_sym_total[i]=rx_sym_tot[i];
786     }
787
788     //Free memory
789     free(pulse_bb);
790     free(rx_in_noisy);
791     free(values);
792     free(help);
793     free(n_SER);
794     free(SER);
795     free(n_BER);
796     free(BER);
797     free(rx_bit_tot);
798     free(rx_sym_tot);
799     free(pulse_mf_inv);
800
801     for(i=0; i<m_psk; i++){
802         free(pulse_mod[i]);
803     }
804     free(pulse_mod);
805
806     for(i=0; i<m_psk; i++){
807         free(mf_out[i]);
808     }
809     free(mf_out);
810
811     for(i=0; i<m_psk; i++){
812         free(pulse_mf[i]);
813     }
814     free(pulse_mf);
815
816     for(i=0; i<5; i++){
817         free(results[i]);
818     }
819     free(results);
820
821     for(i=0; i<16; i++){
822         free(gray_angle[i]);
```

```
823     }
824     free(gray_angle);
825
826 }
```